AFRL-RI-RS-TR-2014-088

A NEW OPERATING SYSTEM FOR SECURITY TAGGED ARCHITECTURE HARDWARE IN SUPPORT OF MULTIPLE INDEPENDENT LEVELS OF SECURITY (MILS) COMPLIANT SYSTEM

UNIVERSITY OF IDAHO

APRIL 2014

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2014-088 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
WILMAR SIFRE
Work Unit Manager

/ S /
MARK H. LINDERMAN
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE	*Form Approved* **OMB No. 0704-0188**

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)* APRIL 2014	**2. REPORT TYPE** FINAL TECHNICAL REPORT	**3. DATES COVERED** *(From - To)* OCT 2010 – OCT 2013

4. TITLE AND SUBTITLE A NEW OPERATING SYSTEM FOR SECURITY TAGGED ARCHITECTURE HARDWARE IN SUPPORT OF MULTIPLE INDEPENDENT LEVELS OF SECURITY (MILS) COMPLIANT SYSTEMS	**5a. CONTRACT NUMBER** FA8750-11-2-0047
	5b. GRANT NUMBER N/A
	5c. PROGRAM ELEMENT NUMBER 62788F
6. AUTHOR(S) Jim Alves-Foss, Jia Song, Stuart Steiner, Saeede Zakeri	**5d. PROJECT NUMBER** T2ST
	5e. TASK NUMBER UN
	5f. WORK UNIT NUMBER ID
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Idaho Center Secure and Dependable Systems 875 Perimeter Drive, MS 1008 Moscow, Idaho 83844-1008	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL/RI
	11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2014-088

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report summarizes the findings of the University of Idaho, Center for Secure and Dependable System's study entitled "A New Operating system for Security Tagged Hardware Architecture in Support of MILS Compliant Systems." The purpose of the project is to investigate the utility of security tagged architectures for high assurance system architectures based on separation and controlled information flow. In addition, this project specifically focused on the use of a zero-kernel operating system as the basis for the evaluation platform for this project.

15. SUBJECT TERMS
Tagged Security Architectures, Operating System Security, MILS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** **WILMAR SIFRE**
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	121	**19b. TELEPHONE NUMBER** (*Include area code*) **N/A**

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

TABLE OF Contents

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This report summarizes the findings of the University of Idaho, Center for Secure and Dependable System's study entitled “A New Operating system for Security Tagged Hardware Architecture in Support of MILS (Multiple Independent Levels of Security) Compliant Systems.” The purpose of the project is to investigate the utility of security tagged architectures for high assurance system architectures based on separation and controlled information flow. In addition, this project specifically focused on the use of a zero-kernel operating system as the basis for the evaluation platform for this project.

This report is divided into four main parts, each of which consists of multiple sections.

Part I of this report consists of the executive summary and background of the project.

Part II presents a discussion of security tagged architectures and zero kernel operating systems. We evaluated several different tagging schemes and summarize their capabilities. We provide a discussion of zero kernel operating systems and unique security schemes of these systems. This section then provides a review of the RTEMS (Real Time Executive for Multiprocessor Systems) operating system which is used as a basis for our evaluation.

Part III presents a discussion of a new security tagging scheme developed as part of this project. The security tagging scheme is supported in both hardware and software, and is designed with the intent of providing separation and controlled information flow. Specifically, this tagging allows isolation of different components of the operating system, implementing a least privilege model.

Part IV of this report consists of a discussion of higher level tagging issues, can we use tagging to support application level security.

1. INTRODUCTION AND EXECUTIVE SUMMARY

The objective of this research was to investigate the design of new operating system architectures in support of the MILS security architecture in the presence of a new security tagged-architecture (STA) microprocessor being developed concurrently by Cornell University research team. The assumption was than an STA-based Operating System (OS) implemented using MILS principles would provide a secure computing foundation that will assist software developers in creating more secure code.

Security tagging schemes are known as promising mechanisms for enhancing the security of computer systems. After studying related work, the advantages of using security tagging schemes are quite clear. Security tagging was first designed and implemented to protect against some low-level attacks, such as buffer overflow and format string attacks [Qin06, shioya09, Yong03, Suh04]. Recently, security tagging schemes have been improved to support prevention of high-level attacks, which can include SQL (Structured Query Language) injection and cross-site scripting [Dalton07, Kannan09]. Tags are also implemented in some architecture to support memory access control [Witchel02, Zeldovich08, Shrobe09, Shriraman10].

Following up on these claims, we proposed to enhance operating system security through the use of these security tags. During this project, the research changed course due to discoveries and refinement of assumptions. The most difficult problem was the lack of a fully functional hardware prototype by the end of the project. The hardware prototype could execute basic security functionality, but could not be automatically reset to continue normal operations after a security exception. This and other findings resulted in a project that focused on the following:

Security Policy Research: The first part of this work focused on investigation into the set of security policies that are enforceable by an STA-based OS running on an STA processor. Prior work in this area defines run-time solutions as meeting a set of enforceable security polices, which are a subset of all policies. We found that a general purpose STA hardware implementation can only enforce a subset of the run-time enforceable security policies, those based on checks for type mismatches. Specifically, the hardware based STA is only aware of the security tags and associated domains that it supports. This mechanism can be used to support higher level security policies similar to how current microprocessors which are not aware of different users can still be used to support separation between those users.

Conclusions: Hardware-based STA can be used to detect type mismatches in memory access, control flow operations and machine code operations. The assignment of types to memory addresses and registers must be under the control of software. Therefore additional software, in the operating system or even at the middleware and application level, is needed to set the tags, and interpret errors generated by the hardware. This software will be more complex than traditional operating system memory protection software and will therefore require increased verification and validation.

STA hardware provides continual checks for type mismatches, relieving software of that burden, and providing greater confidence in the correct behavior of the system. However, care must be taken to not assume more functionality in the STA hardware than really exists. Additional work is needed to understand the tradeoffs between STA supported security features and software-only based security features.

In addition, in order to provide strong assurance of run-time enforceable security policies we contend that all executable hardware of the system (Direct Memory Access (DMA) controllers, co-processors, network cards, etc.) will have to conform to STA principles, or will have to be isolated by STA hardware; a future area of research.

Security Tagging for a Real-Time Operating System: The second part of this project involved research into operating system architectures and architectural support for STA. This involved development of a new tagging scheme and associated security policy, simulation and testing of that tagging scheme, and implementation of an operating system prototype that utilizes the tagging scheme.

It was decided to utilize the RTMES operating system as a basis for the STA-based operating system instead of starting from scratch. Our original focus was on system architecture and theory but we found that working with an operational system allowed us to investigate our solution on a real system, encountering problems that are often abstracted away at the theoretical and architectural levels. Utilizing RTEMS also enabled us to avoid much of the lower level intricacies of a real system and focus on the security aspects of operating system core components.

Conclusions: It is possible to use an STA to provide increased memory protection, separation and isolation in the operating system and among system applications. Enhanced tags can be used to type memory in terms of function entry points, executable code and data. In addition, the tags can be used to limit control flow between different functional units, and limit access to sets of code modules even in the same address space. This is fully in line with the MILS security architecture concept.

Additional work needs to be conducted to determine the best mapping of STA features and tag data types to operating system and application needs. A major drawback of enhanced tagging is the increased overhead of maintaining the tags in memory, and accesses to that memory concurrently with system data.

1.1 PROJECT OVERVIEW

The remainder of this report is organized as follows. Section 2 provides the general background for this work. Section 3.1 provides details on the zero kernel operating system concept and surveys various tagging schemes that have been proposed by other researchers and developers. Section 3.2 gives an overview of RTEMS and its architecture and discusses security problems that are found in RTEMS and proposes the corresponding tagging features that could address the problems. Section 3.3 presents our security tagging scheme. This section defines the format of our tag and illustrates how each field in the tag is used to help secure RTEMS. The information flow rules for C programming language semantics are also discussed. Section 3.4 maps the information flow rules for the C programming language tags to SPARC assembly language instructions. Section 3.5 concludes the work and provides some directions for future work. The remainder of this report provides the background, details of our tagging scheme and a discussion of the research conducted in this project.

Approved for Public Release; Distribution Unlimited.

2. METHODS, ASSUMPTIONS AND PROCEDURES

2.1 INTRODUCTION

This section introduces the concepts of some security tagging schemes and security tagged architecture that have been presented in the literature.

2.1.1 Different security tagging approaches

The idea behind security tagging schemes is to attach labels to memory or registers to carry information about the tagged data. These tags are carried throughout the runtime. They could be used to ensure the semantics of the computation are correctly implemented; they could be used to isolate code and data, users and system; or they could also be used to enforce security policies at the hardware level. The implementation of tagging in hardware provides developers with enhanced security mechanisms with improved performances, as compared to traditional microprocessors. Therefore, tagging schemes are seen as promising mechanisms that help computer systems work properly and securely.

Through an evaluation of the studies involving different security tagging approaches, we have determined that the purpose of using these approaches can be divided into two categories: using security tags for attack prevention and using security tags for access control.

2.1.1.1 Implementing security tags for attack prevention

Some of the security tagging schemes are implemented to prevent low-level attacks, such as buffer overflow and format string attacks. The following is a summary of several proposed tagging schemes, which are presented in more detail in Section 3.1.

In dynamic information flow tracking (DIFT), designed by Suh et al. [Suh04], registers and memory are tagged with a 1-bit tag to indicate spurious data or authentic data. The untrusted inputs are tagged as spurious. Then, by using information flow tracking, all of the information flows from this untrusted data are tracked by the processor. If a dangerous use of a spurious value is found, the processor generates a trap to a software handler for further checks. If the spurious data is not permitted under the enforced security policy, the handler will terminate the application. This approach is focused on securing the system against buffer overflow and format string attacks.

LIFT [Qin06] is a software-based information flow tracking system that uses dynamic binary instrumentation and optimizations to detect attacks. It is said that LIFT can protect against eighteen different buffer overflow attacks [Qin06].

Shioya et al. [shioya09] presented a new security tagged architecture that uses a tag table, a multilevel table, and tag caches to reduce the overhead of security tagged architecture. Compared to the traditional security tagged architecture, the author shows that the new architecture significantly reduces the memory overhead compared to other tagging schemes.

Since it is not enough to only protect against low-level attacks, more schemes are being designed and implemented to prevent high-level attacks, such as SQL injection and cross-site scripting.

According to Yong and Horwitz [Yong03], their approach is a security-enforcement tool for C programs that protects against attacks via unchecked pointer dereference. This tool can identify unsafe pointers and uses a 1-bit tag to indicate appropriate and inappropriate pointer usage. It tags each byte of memory to show whether it is an appropriate location that an unsafe pointer can point to. If the memory location being written or freed is not tagged as appropriate, an error message is raised and the program is halted.

Raksha [Dalton07] is a flexible architecture based on DIFT. In order to detect more attacks, Raksha adds a 4-bit tag to registers, cache and memory locations. Among the 4 bits, 2 bits are used for detecting high-level attacks, 1 bit is for detecting memory corruption and the other 1 bit is for low-overhead security exceptions. Beyond buffer overflow, Dalton [Dalton07] claims that Raksha is able to detect directory traversal, command injection, SQL injection, and cross-site scripting attacks.

Decoupling DIFT [Kannan09] proposes to use a small coprocessor that implements DIFT, that is, it stores register tags, tag caches and also performs the tag propagation and tag checks. The system attaches an asynchronous coprocessor to the main processor core and synchronizes between these two cores when there is a system call.

2.1.1.2 Implementing security tags for access control

In addition to use of tags for attack prevention, researchers have also used tags as security labels for access control and protection of memory and objects.

Mondrian Memory Protection (MMP) [Witchel02], is a fine-grained memory protection scheme that extends the traditional memory protection scheme. Compared with current memory protection, which sets access permissions at the page level, MMP allows arbitrary access control for words.

Loki [Zeldovich08], is a hardware architecture that supports a word-level memory tagging approach. Loki is ported to Histar, an operating system that contains a small trusted kernel. Loki helps reduce the amount of trusted code in Histar. Different from Mondrian memory protection, Loki tags every 32-bits of physical memory with a 32-bit tag. Tag values represent the security policy that should be enforced.

According to Shrobe, DeHon and Knight [Shrobe09], trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA) is a codesign of hardware, operating system, and applications. A zero kernel operating system is designed to run on the security tagged architecture. TIARA implements access controls at different levels. At the hardware level, the tag management unit enforces fine-grained access control for individual words in memory and registers. Higher levels use other access control mechanisms.

Details about each of these approaches are provided in Section 3.1.3

2.1.2 Security tagged architecture

The concept of security tagged architecture was first proposed by Feustel in 1973 [feustel73]. However, it has not received significant attention until recent years. As computer hardware has been getting faster and more functionality is now able to be put on chips, it is becoming easier to use hardware to provide enhanced functionality normally implemented in software. A security tagged architecture is an architecture that supports tagging capabilities such as tag storage, tag propagation, and tag checking in hardware. Therefore, by using security tagged architecture, the maintenance of correctness and security of the system can be performed at the hardware level. This will reduce the consumption of system resources, provide a common platform for security, and reduce the possibility of bypassing or temporizing with the security mechanism.

The architecture, defined by Shrobe, DeHon and Knight [Shrobe09], is a hardware architecture that allows metadata to be used during runtime to enforce security policies and to maintain the semantics of the computation. The metadata can be represented with a tag. It is usually associated with a word of memory and carries information about the word, such as access rules, permissions, data types, and so on. There have been a few security tagged architectures designed to support tagging schemes. Four of the more notable security tagged architectures are discussed below.

2.1.2.1 Mondrian memory protection architecture

The architecture of Mondrian memory protection (MMP) is shown in Figure 1 (adapted from Witchel, Cates and Asanovic [Witchel02]). In MMP, every allocated memory region is owned by a protection domain. Each running thread has a protection domain ID and each domain has a corresponding permission table. The permission table specifies the permission of each address that the domain has. The permission table is stored in memory.

Every access to memory requires a check of the permission table. The processor checks the permissions in the sidecars. Sidecars are registers that hold information for one table segment. Each address register has one sidecar register associated with it that caches the last table segment accessed by this address register. If the sidecars do not have the permissions information, the processor will then reload the sidecar from permission lookaside buffer (PLB). If the needed permission information needed is not in PLB, the permission information will be retrieved from the permissions table resident in memory, by means of the hardware or software, and refilled into both PLB and sidecars.

Approved for Public Release; Distribution Unlimited.

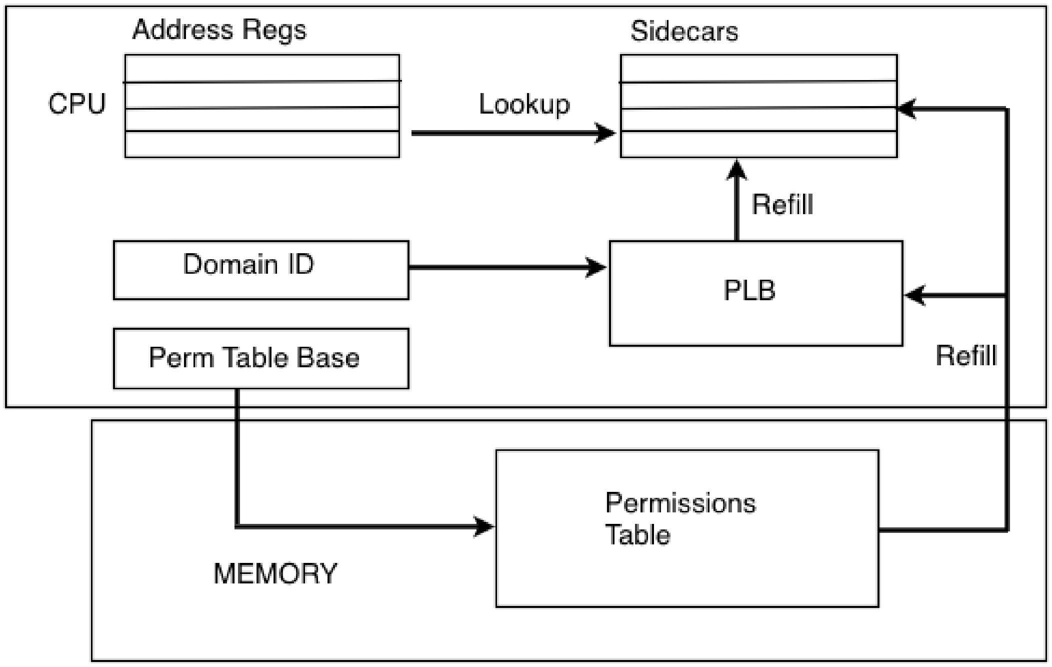


Figure 1: Mondrian Memory Protection Architecture

2.1.2.2 Dynamic information flow tracking architecture

DIFT [Suh04] implements the security tag scheme in a processor. Shown in Figure 2 (adapted from Suh et al. [Suh04]), the on-chip architecture has new structures (ITag TLB, DTag TLB, T$-L1, T$-L2) that support the tagging scheme. In DIFT, each register has an additional bit to store the tag. DIFT also adds two separate tag caches, T$-L1 and T$-L2. Because of the introduction of the new tag caches, a new translation table between L2 cache and memory is needed. DIFT has ITag TLB and DTag TLB for security tags. When accessing memory, the ITag TLB returns information about the tag type and the DTag TLB returns pointers which indicate the location of the tags.

DIFT has two new registers: propagation control register (PCR) and trap control register (TCR). PCR has a bit vector that controls the propagation of tags based on the security policies. TCR has information about whether or not to generate a trap for a specific kind of values. Both registers are configured by the execution monitor, which is a software module used to enforce the security policies. The PCR and TCR in the security tagged architecture provide DIFT with the ability to evaluate and propagate tags at the hardware level and throw an exception to the software level when an error occurs.

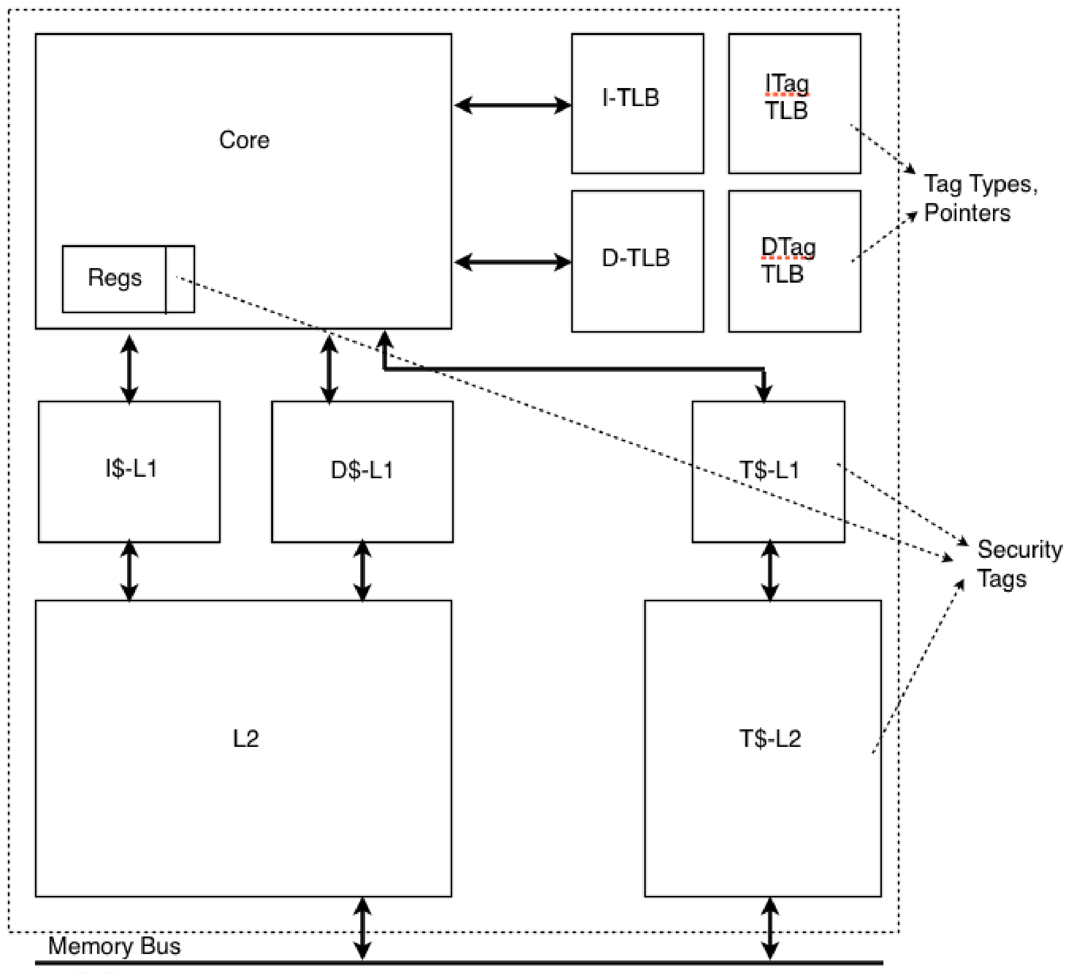


Figure 2: DIFT hardware architecture

2.1.2.3 Dedicated Coprocessor architecture

Kannan, Dalton, and Kozyrakis [Kannan09] proposed a decoupling dynamic information flow tracking architecture based on a dedicated small coprocessor. Compared with using a separate general core, the small coprocessor is a better choice because it is sufficient for doing tag propagation, checks, and does not impact normal CPU (central processing unit) operations. DIFT functionality is implemented in the small attached coprocessor. The main processor core will be synchronized with the small coprocessor when doing system calls. As Figure 3 shows (adapted from Kannan, Dalton and Kozyrakis [Kannan09]), the small attached coprocessor includes a tag pipeline for tag propagation and a tag cache to store and maintain the tags.

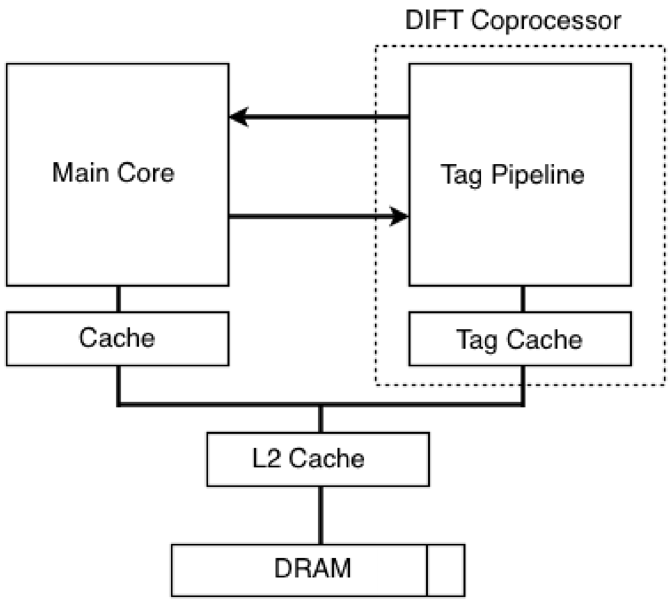


Figure 3: Dedicated coprocessor architecture

According to Kannan, Dalton, and Kozyrakis [Kannan09], this off-core coprocessor design reduces the complexity of DIFT support in the hardware. The main processor core passes information about the information flow to the coprocessor, then the coprocessor performs all of the tag propagation and maintains the tag state. Therefore, the main processor does not know about the existence of tags[1].

2.1.2.4 TIARA architecture

Figure 4 (adapted from Shrobe, DeHon, and Knight [Shrobe09]) shows the Tag Management Unit (TMU) used in TIARA and also the tagged data path. In TIARA, the TMU runs parallel with the data path. It operates on the tag of the data and makes decisions on whether or not to send a trap signal based on the enforced security policies. The TMU takes certain fields of the operands, the program counter, and the instruction to compose the tags for the data. In TIARA, every word in memory and all of the processor registers are tagged to show their data type and security context information. In addition, the program counter is tagged, which helps handle conditional branch instructions. TIARA implements a policy table in the main memory. Similar to a regular cache, if a miss occurs in TMU, it will look up the policy table and load the proper entry into its memory.

[1] It is this coprocessing scheme that our partner hardware development team, led by Edward Suh at Cornell University, is now exploring.

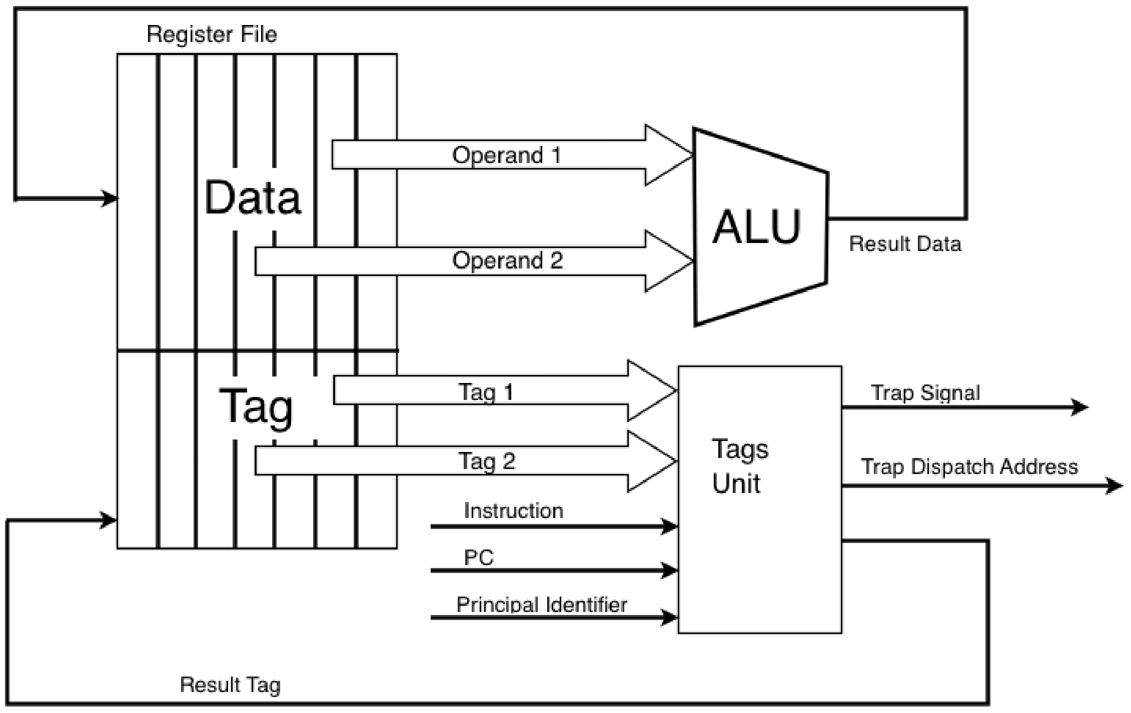


Figure 4: TIARA TMU architecture

According to Shrobe, DeHon, and Knight [Shrobe09], when the main data path is executing an instruction, the TMU deals with the tags of the two operands, the program counter's tag, the instruction, and the principal register which stores the privileges of the running process. If the execution of the instruction does not violate the access control policies, the resulting tag is derived from the tags of the input operands. However, if the execution violates the security policies, then the tag unit forces the process to the trap handler.

2.2 SECURITY PROBLEMS

Due to the increasing reliance on technology, wide varieties of software are installed on computers, smart phones, tablets and other computing devices. As attacks can exploit the vulnerabilities either in operating systems or in the installed software, security issues of computer systems has become one of the biggest concerns in people's lives. This section summarizes some common security attacks in today's computing environments. They can be addressed by our proposed tagging scheme, and some can also be addressed by other tagging schemes found in the literature.

Memory corruption attack. A memory corruption attack causes the system to violate the standard programming language semantics by overwriting unexpected areas of memory. The buffer overflow attack is the most common type of memory corruption attack. If the data written to a buffer exceeds the capacity of the buffer, the extra data will overwrite the memory space

Approved for Public Release; Distribution Unlimited.

adjacent to the allocated buffer. The attack stems from insufficient bounds checking. If the memory where the return address is stored is overwritten, then once the function returns, execution will resume at the address specified by the attacker which can force execution of malicious code. This kind of attack can be prevented by security tagging schemes such as DIFT [Suh04], Raksha [Dalton07], LIFT [Qin06], Low overhead architecture for security tag [shioya09], Invalid pointer dereference tool [Yong03], Decoupling dynamic information flow tracking [Kannan09] and our tagging scheme.

SQL injection. SQL injection is used to attack a website by sending malicious SQL statements to it, so as to get the defective website to release private contents of the database to the attacker. This attack is due to the use of badly designed query language interpreters. When a user input is incorrectly filtered or not strongly typed, malicious SQL commands sent to the website's software can be used to exploit security vulnerabilities of the website. According to their authors, SQL injection can be prevented by using tagging schemes like Raksha [Dalton07] and Decoupling dynamic information flow tracking [Kannan09].

Data isolation. Attackers may forge pointer values to access the contents of unauthorized memory space, such as critical memory blocks owned by the operating system. The attackers can then use this access to read or write the memory space pointed to by the forged pointer. If the system does not check the validity of the pointer and allows the attacker to write the value of the memory space, then the attacker can have complete control of the system. This attack can be prevented by our scheme, Invalid pointer dereference tool [Yong03], and memory access controls such as those found in TIARA [Shrobe09].

Forged index or reference value. Most operating systems manage a large number of resources for multiple users. System calls used to access these resources are usually passed through a reference or index to indicate which resource the user wishes to access. It might be possible for an attacker to forge an index which refers to a resource not owned by the attacker. If the system does not validate that the reference belongs to the user, the attacker could force the system to corrupt, delete, or release contents of this resource. To prevent this kind of attack, access controls are need. This attack can be prevented by our tagging scheme (Section 3.3).

2.3 MOTIVATION AND CONTRIBUTIONS

The main part of this project was focused on the development of a new security tagging scheme which could be implemented in an operating system to support a new security tagged architecture microprocessor. The objectives of this project are as follows:

Objective 1: Survey existing tagging schemes. Since many different security tagging schemes have been proposed and implemented, the survey of these schemes may help us understand the current scope of tagging schemes and provide insight into how to design our tagging scheme.

Objective 2: Evaluate the RTEMS source code to find potential security problems. RTEMS is a set of runtime executives which provide different functionalities. Reading the source code of RTEMS helps us understand its architecture and how the code works. We are

Approved for Public Release; Distribution Unlimited.

designing a tagging scheme that will be implemented as an extension of RTEMS. Therefore, we must have a thorough understanding of RTEMS.

Objective 3: Develop a new security tagging scheme for RTEMS. This is the main purpose of this project. We designed a new security tagging scheme that can address the current security problems in RTEMS while using as much hardware support as possible to keep performance overhead low.

Objective 4: Apply the tagging rules to SPARC (Scalable Processor ARChitecture) instructions at the assembly language level. Our new security tagging scheme will be eventually implemented in the SPARC architecture. Therefore, we need to ensure that our design is able to be implemented on the hardware architecture. Reviewing the assembly code generated by GNU Compiler Collection (GCC), we were able to see that a C-Language level tagging definition is incomplete and that we needed to refine the definition to enable the hardware implementation.

The design presented in this project is a new idea for using tags to enforce the security of an operating system. The new security tagging scheme proposed is designed for the RTEMS runtime operating system to address the security issues of RTEMS. We examined the RTEMS source code to evaluate potential security problems and designed tags to fix these problems for the system. In our security design we use tags to separate RTEMS code and data from user code and data. By using tags, we also propose security mechanisms to control the access to system calls and control information flow within RTEMS. Lastly, we studied the SPARC architecture and refined the security rules that we proposed for the C programming language at software level to SPARC assembly language instructions.

In summary, the new security tagging scheme presented in this project provides a way to protect the system and user software from security attacks and invalid information access by using tags. By being implemented in hardware, we achieve a performance increase as well as a common security mechanism to support all of the code, alleviating the OS (Operating System) from the responsibility of ensuring that all OS code correctly implements security features.

Approved for Public Release; Distribution Unlimited.

3. RESULTS AND DISCUSSION

This section describes the results of our research organized in the following manner:
● Security Tagged Architectures and Related Technologies
● Zero Kernel Operating System and RTEMS
● New Security Tagging Scheme
● Implementation of Tagging Scheme at the Assembly-Language Level
● Extension and Evaluation of the Tagging Scheme
● Tagging and Security Policies
● Application Tagging

3.1 SECURITY TAGGED ARCHITECTURES AND RELATED TECHNOLOGIES

3.1.1 Background

Section 3.1 provides background on the zero kernel operating system concepts and surveys various tagging schemes.

Shrobe, DeHon, and Knight [Shrobe09] proposed the concept of a zero-kernel operating system (ZKOS). The basic idea of a ZKOS is to decompose the whole system into many smaller components. Each component maintains its own compartment and principal. A compartment defines the resources and memory used by the compartment and the principal defines the subject (or user) that own the compartment. The operating system components run as part of user code, like library routines, but with special protections provided by a tagging mechanism. The details are discussed in Section 3.1.2.

From our study of related research for tagging schemes, we find that tagging has recently been used in two major areas, protecting against security attacks and to control memory access. The tagging schemes for attack prevention are Dynamic information flow tracking [Suh04], Raksha [Dalton07], LIFT [Qin06], Low overhead architecture for security tag [shioya09], Invalid pointer dereference tool [Yong03] and Decoupling dynamic information flow tracking [Kannan09]. Tagging schemes for access control include Mondrian memory protection [Witchel02], Sentry [Shriraman10], Loki [Zeldovich08] and TIARA [Shrobe09]. The details of these tagging schemes are discussed in Section 3.1. Some of the schemes are implemented only at the software level, while some of them need support from the hardware level, which requires a design of tagging engines and changes to the hardware.

In Section 3.1, the idea of ZKOS is introduced in Section 3.1.2. Section 3.1.3 discusses the tagging schemes for attack prevention. The tagging schemes for access control are discussed in Section 3.1.4. The last section, Section 3.1.5, summarizes all of the tagging schemes that we surveyed.

3.1.2 Zero Kernel Operating System

A good operating system can protect system code from modification by user or software. Conventionally, as shown in Figure 5, to satisfy this requirement systems separate memory into user space and kernel space. To further control separation, systems support the concept of virtual memory, which allows user code to run unmodified as the operating system reallocates memory regions or even swaps memory out of RAM to disk. Programs running in kernel space have ultimate privileges, while programs running in user space have only limited privileges. To be able to support the separation of user space and kernel space, separate kernel and user execution modes are traditionally required. However, the transition between kernel mode and user mode, or between different users spaces, namely the *context switch*, incurs a huge performance overhead, as registers need to be saved and restored and memory maps need to be reloaded. In addition, the current implementations of virtual memory cause multiple memory accesses upon page misses.

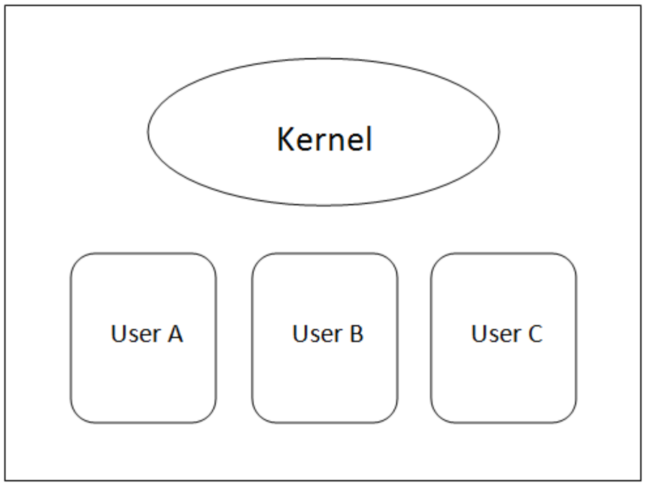


Figure 5: Traditional operating system model

Shrobe, DeHon, and Knight [Shrobe09], designed a new operating system called a zero-kernel operating system (ZKOS) to address performance and security issues of current operating systems. Along with the support of a new security tagged architecture, ZKOS provides fine-grained memory protection by using tags. As shown in Figure 6, ZKOS decomposes the whole operating system into many smaller components based on an object-oriented design philosophy. Each component has its own compartment and principal. A compartment defines the resources and memory used by the compartment to store data and the principal specifies the authority of operations. A component's principal can access the data of its own compartment, but cannot access other components' data and user's data. If data needs to be shared among two or more components, then shared compartments are used to support it.

Traditional memory protection, protected mode, and supervisor mode gives the programs running in supervisor mode ultimate privileges and the programs running in protected mode limited privileges. Different from the protected mode and supervisor mode, ZKOS decomposes the whole operating system into many smaller components, each having its own limited privileges. This ensures that the component has the least privileges needed. ZKOS also prevents any of the components from having complete authority. Therefore, even if a component is compromised, the whole system will not be compromised. By using smaller components, the system has better isolation in code and data and also a better separation of system and users. Supported by a security tagged architecture, ZKOS avoids the expensive context switches between kernel mode and user mode that occur in traditional operating systems because the components have their own privileges and compartments. What is more, the decomposed components help ensure fine-grained memory protection of the system.

Shrobe, DeHon, and Knight [Shrobe09] require that each component consist of not only compartments and principals, but a set of access rules as well. Access rules are used to restrict access from other components that do not have the appropriate privileges. ZKOS is designed to give each component only the minimal privileges that are needed. Moreover, the operating system components are unable to access data in other operating system compartments and even the user's compartments. The interaction and sharing among different ZKOS components are also well structured. In simple terms, each component establishes a set of gates that include a

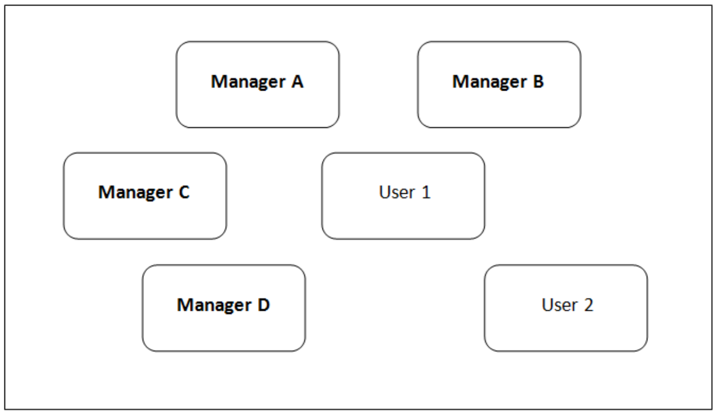


Figure 6: Zero kernel operating system model

procedure, a compartment, and a principal to indicate the entry points of the service. With the help of the TMU (discussed in Section 2.1.2.4) the invocation of the gate is only allowed by the components that are authorized.

In summary, compared with a traditional operating system, ZKOS allows better separation of users and the system and a better separation of system modules. It also avoids the expense of performing context switches between kernel mode and user mode. The components turn the system into a multilevel system, which helps provide better control of information flow. With the support of an enhanced security tagged architecture, a tagged ZKOS would be more flexible and powerful.

3.1.3 Tagging Approaches for attack prevention

Tagging schemes can be categorized into two classes: using tags for attack prevention and using tags for access control. This section provides more detailed information about tagging schemes for attack prevention summarized in Section 1. These schemes include Dynamic information flow tracking [Suh04], Raksha [Dalton07], LIFT [Qin06], Low overhead architecture for security tag [shioya09], Invalid pointer dereference tool [Yong03] and Decoupling dynamic information flow tracking [Kannan09].

3.1.3.1 Dynamic Information Flow Tracking

Dynamic information flow tracking (DIFT), proposed by Suh et al. [Suh04], protects programs by tracking input data from untrusted I/O input. A software module in the operating system initializes untrusted I/O input as spurious data. All of the information that flows from the untrusted data is then tracked by the processor. If a dangerous use of a spurious value is found, the processor generates a trap to a software handler to deal with it. Thus, the illegal use of data can be terminated.

Figure 7 (adapted from Suh et al. [Suh04]), gives an overview of the DIFT system architecture. As shown in Figure 7, the implementation of DIFT has three major parts: the execution monitor, the tagging units, and the security policy. The execution monitor tracks information flows and generates traps when spurious data is involved in certain kinds of operations. The trap handler then checks the spurious data and the operation. If the operation is not permitted by the security policy, the handler terminates the application. Otherwise it returns to the application.

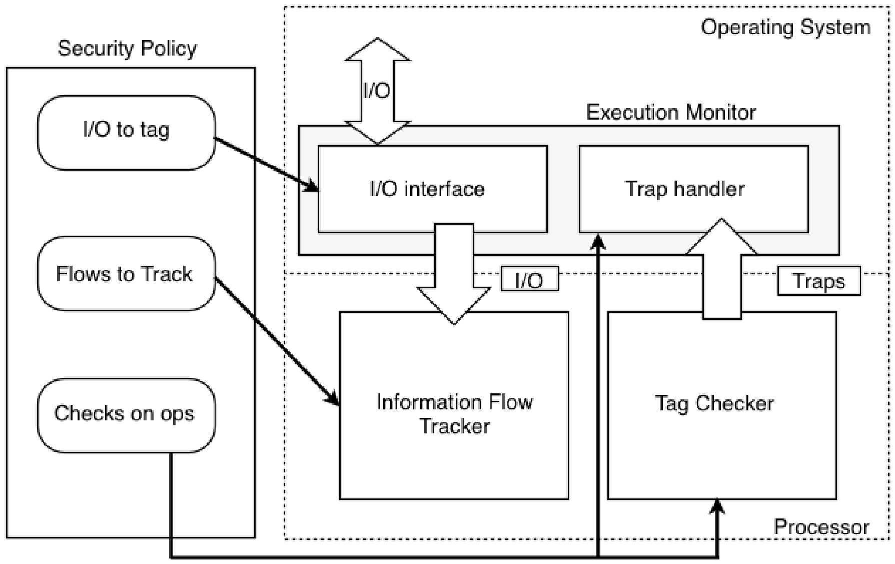


Figure 7: DIFT system architecture

In this approach, a 1-bit tag is attached to each register and each byte in memory. Zero indicates authentic data and a one indicates spurious data. At the beginning, all tags are initialized to zero. According to Suh [Suh04], DIFT tags all I/O input, except the initial program loaded from disk. When executing an instruction, the processor checks the tag of each operand.

When it encounters a spurious value, the tag with this block of data will be changed to a one to indicate dangerous data using the following rules:

Copy dependency: The value copied from a spurious value is also a spurious value.

Computation dependency: The value computed from a spurious value is also a spurious value.

Load-address dependency: A value loaded from a memory address that is spurious is also a spurious value.
Store-address dependency: A value stored to a memory address that is spurious becomes a spurious value.

Control dependency: If the execution path depends on a spurious value, then all of the program states are spurious.

DIFT uses security policies to specify the untrusted I/O input, to track the information flows, and to handle the trapped value. According to Suh [Suh04], the general security policy is “No instruction can be generated from I/O inputs, and no I/O inputs and spurious values based on propagated inputs can be used as pointers unless they are bound-checked and added to an authentic pointer.”

DIFT works well for detecting buffer overflows and format string attacks. The performance evaluation of DIFT in Suh's paper [Suh04] indicates that DIFT, on average, has a 1.44% memory overhead and a 1.1% performance degradation[2].

3.1.3.2 Raksha

According to Dalton, Kannan, and Kozyrakis [Dalton07], Raksha is a flexible architecture based on DIFT. It uses information flow tracking to enforce software security. Since DIFT only uses one security policy, which focuses on preventing memory corruption attacks, it cannot detect other attacks such as SQL injection, cross-site scripting, and command injections. In order to detect more attacks, Raksha adds a 4-bit tag to registers, cache, and memory locations. Among the 4 bits, 2 bits are used for high-level attacks, 1 bit is used to detect memory corruption and the other 1 bit is for low-overhead security exceptions.

Raksha provides three major features. First, it supports multiple security policies, which allows Raksha to protect against more attacks. Beyond buffer overflow, Dalton, Kannan, and Kozyrakis claim that Raksha is also able to detect directory traversal, command injection, SQL injection, and cross-site scripting attacks. Second, the security policies enforced by Raksha are flexible and programmable. Raksha implements two new registers: tag propagation registers (TPR) and tag check registers (TCR). Each security policy is supported by a pair of TPR and

[2] We worked with Suh as part of this project, to extend his architecture.

TCR. By configuring these registers, Raksha can support different security policies. Third, security exceptions can be handled at the user-level. Because of the high cost of generating an OS trap, Dalton, Kannan, and Kozyrakis have designed Raksha to handle security exceptions at the user-level. However, this requires protection of the code and data used by the exception handler. To support user-level exception handling, apart from user mode and kernel mode, Raksha implements a new mode, trusted mode, for the processor. When a security exception is raised, the processor enters trusted mode to handle the exception.

3.1.3.3 LIFT

According to Qin et al. [Qin06], LIFT is an information flow tracking system that uses dynamic binary instrumentation and optimizations to detect attacks. LIFT is a software only system, thus it requires no hardware extensions. It uses a 1-bit tag to tag the data in memory or general data registers. Similar to DIFT, a one indicates unsafe data and a zero indicates safe data. LIFT introduces three ways of optimization:

Fast Path Optimization. There will be a simple check of the data in the execution region before performing the execution. If there is no unsafe data involved, the dynamic information flow tracking will not be performed for this computation. In this case, unnecessary dynamic information flow tracking can be eliminated. Therefore, the information flow track overhead is reduced, since most of the computations only involve safe data.

Merged Check Optimization. LIFT merges multiple tag checks of consecutive memory locations into one check to reduce the check overhead.

Fast Switch Optimization. LIFT uses condition register liveness analysis to discard meaningless condition register save and restore. It uses some simpler instructions instead of more expensive instructions, which reduces the runtime overhead.

LIFT stores the tags for memory data in tag space, which is a special memory region. According to Qin et al. [Qin06], this tag space incurs 12.5% space overhead. Since tags for registers are accessed frequently, the tags for registers are stored in a 64-bit register to enable faster access.

Because the LIFT code and tag space could be corrupted by malicious programs, LIFT provides methods to protect them. To protect the LIFT source code, page protection is used to mark the memory pages that store the LIFT code as read-only. The tag space is protected by turning off the access permission of the specific memory pages that store the tag value of the tag space. By doing this, any modification to LIFT's code results in a page fault, and any attempt to access the value in the tag space causes a protection fault.

Qin et al. indicate that LIFT can protect against 18 different types of buffer overflow attacks with low overhead [Qin06], 6.2% for server applications.

3.1.3.4 Low overhead architecture for security tag

Shioya et al. [shioya09] presented a new security tagged architecture that uses a tag table, a multilevel table, and tag caches to reduce the overhead of the security tagged architecture.

Compared to the traditional security tagged architecture, the authors show that the new architecture significantly reduces memory overhead. The memory used for tags takes no more than 1.8%, on average, of the memory not used for tags.

This architecture exploits two features of tags, non-uniformity and locality of reference. Non-uniformity indicates that some of the memory has tags associated with it and the others do not need to be tagged. Locality of reference means that tags are used locally in code similar to data. Therefore tags are cached more effectively.

3.1.3.5 Invalid pointer dereference tool

Yong and Horwitz's [Yong03] invalid pointer dereference tool is a security-enforcement tool for C programs which protects against attacks via unchecked pointer dereference. This tool can identify unsafe pointers and uses a 1-bit tag that tags memory to indicate it is an appropriate or inappropriate memory location to be referenced by an unsafe pointer. The 1-bit tag is associated with each byte of memory. If the memory location being written or freed is not tagged appropriately, this means that the unsafe pointer cannot point to this memory location. Thus, an error message is raised and the program is halted.

This tool first performs a “points-to” analysis to determine the proper memory location that a certain variable may point to during execution of the program. In this step, all of the variables are analyzed. Then the tool identifies unsafe pointers. According to Yong and Horwitz [Yong03], a pointer will be identified as an unsafe pointer when (1) the pointer refers to an invalid memory location at runtime and (2) the pointer is dereferenced for writing, or the free function is called. Lastly, the tool will check tags during runtime and stop illegal actions.

3.1.3.6 Decoupling dynamic information flow tracking

Decoupling dynamic information flow tracking (Decoupling DIFT), presented by Kannan, Dalton, and Kozyrakis [Kannan09], checks tags associated with the instruction's memory location. This scheme is focused on making DIFT [Suh04] more efficient by changing the hardware.

Kannan, Dalton, and Kozyrakis [Kannan09] claim that in the DIFT architecture, tag propagation and checks are performed in the processor pipeline, which requires minimal overhead. However, a lot of work must be done modifying the processor core because of the fact that pipeline stages, registers, and caches need to be changed to maintain the tags. Decoupling DIFT proposes to use a small coprocessor that implements DIFT, by storing register tags, tag caches, and also performing the tag propagation and tag checks. Systems attach the coprocessor to the main processor core and synchronize between these two cores when system calls happen.

The authors [Kannan09] mention that, compared with the common version of DIFT, Decoupling dynamic information flow tracking scheme reduces the performance overhead from 7% to less than 1%.

Approved for Public Release; Distribution Unlimited.

3.1.4 Tagging approaches for access control

The previous section surveyed tagging schemes for attack prevention and information flow control. Tags have also been used for access control. Security tagging schemes for access control include Mondrian memory protection [Witchel02], Sentry [Shriraman10], Loki [Zeldovich08] and TIARA [Shrobe09]. In this section, these tagging schemes are discussed in detail.

3.1.4.1 Mondrian memory protection

According to Witchel, Cates, and Asanovic [Witchel02], Mondrian memory protection (MMP) is a fine-grained memory protection scheme that extends the traditional memory protection scheme. Compared with the memory protection, which sets access permissions at the page level, MMP allows arbitrary access control for individual words. Mondrian memory protection uses two permission bits to support different access control for individual words. These 2-bit permission bits present four meanings: no permission, read-only, read/write, and execute/read.

Witchel et al. [Witchel02] states that MMP uses compressed permissions table and two levels of permissions caching to lower the space overheads and performance overheads.

3.1.4.2 Sentry

Sentry, designed by Shriraman and Dwarkadas [Shriraman10], is a virtual memory tagging scheme that provides lightweight auxiliary memory access control that can be controlled at the user level. Sentry implements a permission metadata cache (M-cache) to intercept L1 misses. The metadata cache also controls whether data is allowed to be stored into the L1 cache. Since data stored in L1 has been checked, access to L1 cache does not need any additional checks. Sentry uses 2-bit permission metadata to indicate the access permission. Each entry in the metadata cache has 2-bit permission metadata to represent access permission to a specific virtual page. The 2-bit permission metadata represents four different access permissions: read-only, read/write, no access, and execute.

3.1.4.3 Loki

Loki, presented by Zeldovich [Zeldovich08], is a hardware architecture that supports a word-level memory tagging approach. Loki is ported to Histar, an operating system that has a small trusted kernel. Loki helps reduce the amount of trusted code in Histar. Different from Mondrian memory protection, Loki tags every 32-bits of physical memory with a 32-bit tag. Tag values represent the security policy that should be enforced by the data. Loki implements a permissions cache, which is used to hold the permission bits for some tags. To avoid the high overhead incurred by using large tags, Loki implements an efficient multi-granular tagging scheme. It allows a page of memory to be tagged with only one 32-bit tag.

According to Zeldovich [Zeldovich08], by using a permission cache (P-cache), Loki enforces fine-grained permission checks. This cache stores the security tag and a 3-bit permission, which represents read, write, and execute. For every memory access, this 3-bit permission is checked to see if the memory location is appropriate to be accessed.

To protect the tag values and the permission cache, Loki implements a new privilege mode, called monitor mode. Only in this monitor mode can tag values and entries of permission cache be changed.

3.1.4.4 TIARA

According to Shrobe, DeHon, and Knight [Shrobe09], Trust-management, intrusion-tolerance, accountability, and reconstitution architecture (TIARA) is a codesign of hardware, operating system, and applications. As discussed in Section 3.1.2, a ZKOS is designed to run on security tagged architecture. According to Shrobe, DeHon, and Knight [Shrobe09], TIARA implements access control at different levels. At the hardware level, the tag management unit (TMU) enforces fine-grained access control for individual words in memory and registers. At a higher level, another access control mechanism is imposed. TIARA modified the traditional access control model to fit in its architecture and operating system. The access control model includes six major parts:

Objects. An object is an instance of a class.

Compartment. A compartment is a collection of objects that have identical access permissions.

Principals. A principal represents a live entity in the system. It could be a user or a component.

Threads. Every thread has a principal and a compartment associated with it.

Access rules. Rules for controlling the access abilities of principals.

Gates. A gate is a set of a generic function, a compartment, and a principal.

Different from the traditional access control methods, which specify who can do what to objects, TIARA implements four classes of access rules to focus on four different aspects:

Controls of read and write to objects. The access rules specify which principals are allowed to read or write a specific object.

Controls of allocation of objects. This class of access rules limits which principals have the ability to allocate an object.

Controls of invocation of gates. This kind of access rules specifies which principals are allowed to invoke which gates.

Controls of invocation of services. The access rules in this class focuses on limiting principals to invoke particular services.

With the tags added in TIARA and enforced by TMU, all the objects within the memory of TIARA are controlled under the access rules.

3.1.5 Summary of tagging schemes

Table 1 shows the attacks that can be prevented by each of these reviewed tagging approaches; it shows if it is a tagging scheme for access control and if it has hardware support. This table shows which attacks can be prevented by using DIFT [Suh04], Raksha [Dalton07], LIFT [Qin06], Low overhead architecture for security tag [shioya09], Invalid pointer dereference tool [Yong03] and Decoupling dynamic information flow tracking [Kannan09]. The Y's in the access control column for Mondrian memory protection [Witchel02], Sentry [Shriraman10], Loki [Zeldovich08] and TIARA [Shrobe09] indicate that these four tagging schemes support access control. Some of the schemes are implemented only at the software level and some need support from hardware level, which requires design of tagging engines and changes to the hardware. We have shown whether or not these schemes need hardware support in the last column.

Table 1: Summary of tagging schemes

	Buffer overflow	Format string	Command injection	SQL injection	Cross-site scripting	Access control	Hardware support
DIFT [Suh04]	Y	Y					Y
Raksha [Dalton07]	Y	Y	Y	Y	Y		Y
LIFT [Qin06]	Y						
Dereference pointers [Yong03]	Y						
Decoupling DIFT [Kannan09]	Y	Y	Y	Y			Y
Low-overhead tagging [shioya09]	Y	Y					Y
MMP [Witchel02]						Y	Y
Sentry [Shriraman10]						Y	Y
Loki [Zeldovich08]						Y	Y
TIARA [Shrobe09]						Y	Y
UI Tagging	Y	Y				Y	Y

Approved for Public Release; Distribution Unlimited.

3.2 ZERO KERNEL OPERATING SYSTEM AND RTEMS

The goal of this project is to design and evaluate a security tagging mechanism for implementation in an operating system running on security tagged architecture. RTEMS (Real-Time Executive for Multiprocessor Systems) [Rtems] is a real-time executive that has 18 managers, each of which can be viewed as an individual module. Since the architecture of RTEMS is very similar to the idea of ZKOS, we decided to use RTEMS to implement our new security tagging scheme. Because RTEMS has almost no protection, the purpose of the new tagging scheme is to secure RTEMS without requiring a major rewrite and reduce the overhead incurred by implementing the tags in hardware. This project focuses on the design and evaluation of the tagging scheme in the context of RTEMS, but leaves implementation to future work.

Section 3.2 first gives an overview of RTEMS and its architecture. Then, discusses problems found in RTEMS and provides the corresponding tagging features that could solve the problems. After examining the RTEMS source code, we found RTEMS is a collection of executives that have no security mechanisms implemented. Therefore, we use example source code to point out some major problems in RTEMS. The tagging scheme is designed to mitigate these security problems.

The remainder of Section 3.2 is organized as follows: In Section 3.2.1, the architecture and important data structures of RTEMS are introduced. Section 3.2.2 discusses the problems we found in RTEMS that may cause security vulnerabilities. The details of our tagging scheme will be illustrated in Sections 3.3 through 3.7.

3.2.1 RTEMS and its architecture

RTEMS [Rtems] is a real-time executive that provides a powerful runtime environment to allow various types of services and applications to be embedded. It can perform a complex set of services that includes multitasking, inter-task communication, and dynamic memory allocation. The bulk of RTEMS is written in both the Ada and C programming languages, which makes it easier to be modified and ported to other processor families, such as ARM, MIPS, PowerPC, SPARC, Atmel AVR etc. In this project, we focus on the C-Language implementation.

As shown in Figure 8: RTEMS Architecture, RTEMS consists of super core (SCORE) (Section 3.2.1.2) and 18 managers (we add two more managers, Tag manager and User manager, to support the tagging scheme, see Section 3.2.1.1). The 18 managers provide different services and the SCORE provides kernel functions that support the managers.

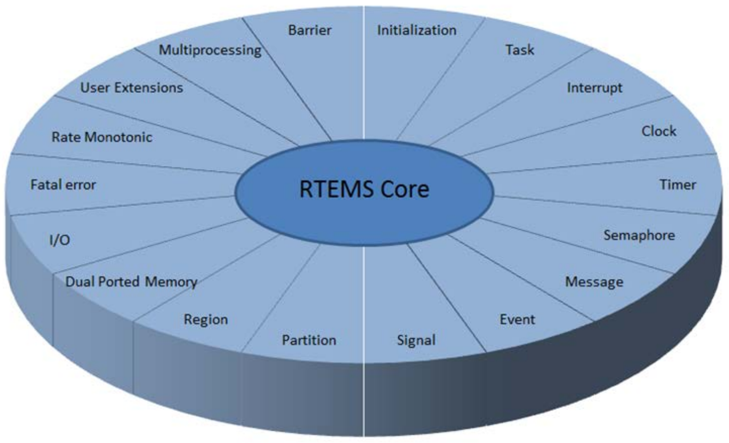


Figure 8: RTEMS Architecture

3.2.1.1 RTEMS managers

RTEMS currently has 18 managers:

- **Initialization manager** - responsible for initializing RTEMS and shutting it down.
- **Task manager** - provides abilities to create, delete, and control tasks.
- **Interrupt manager** - facilities to quickly respond to the external interrupts.
- **Clock manager** - deals with the current date and time issues.
- **Timer manager** - provides facilities to generate and configure a timer.
- **Semaphore manager** - provides synchronization and mutual exclusion capabilities.
- **Message manager** - uses messages to allow communication and synchronization between tasks.
- **Event manager** - provides a mechanism for inter-task communication and synchronization.
- **Signal manager** - uses signal sets to allow asynchronous communication between tasks.
- **Partition manager** - dynamically allocates memory in fixed-sized units.
- **Region manager** - dynamically allocates memory in variable sized units.
- **Dual Ported Memory manager** - helps in converting addresses between internal and external representations for multiple dual ported memory areas.
- **I/O manager** - facilities to access and organize device drivers.
- **Fatal Error manager** - processes all errors.
- **Rate Monotonic manager** - facilities to manage periodic tasks.
- **User Extensions manager** - provides a mechanism to invoke user-defined routines.
- **Multiprocessing manager** - helps provide real-time multiprocessing capabilities.
- **Barrier manager** - provides a unique synchronization capability.

Table 2: Directives of each manager

Manager	Number of Manager	Directives
Initialization	5	rtems_initialize_data_structures, rtems_initialize_before_drivers, rtems_initialize_device_drivers, rtems_shutdown_executive, rtems_initialize_start_multitasking
Task	19	rtems_task_create, rtems_task_ident, rtems_task_self, rtems_task_start, rtems_task_restart, rtems_task_delete, tems_task_suspend, rtems_task_resume, rtems_task_is_suspended, rtems_task_set_priority, rtems_task_mode, rtems_task_get_note, rtems_task_set_note, rtems_task_wake_after, rtems_task_wake_when, rtems_iterate_over_all_threads, rtems_task_variable_add, rtems_task_variable_get, rtems_task_variable_delete.
Interrupt	5	rtems_interrupt_catch, rtems_interrupt_disable, rtems_interrupt_enable, rtems_interrupt_flash, rtems_interrupt_is_in_progress.
Clock	10	rtems_clock_set, rtems_clock_get, rtems_clock_get_tod, rtems_clock_get_tod_timeval, rtems_clock_get_seconds_since_epoch, rtems_clock_get_ticks_per_second, rtems_clock_get_ticks_since_boot, rtems_clock_get_uptime, rtems_clock_set_nanoseconds_extension, rtems_clock_tick.
Timer	10	rtems_timer_create, rtems_timer_ident, rtems_timer_cancel, rtems_timer_delete, rtems_timer_fire_after, rtems_timer_fire_when, rtems_timer_initiate_server, rtems_timer_server_fire_after, rtems_timer_server_fire_when, rtems_timer_reset.
Semaphore	6	rtems_semaphore_create, rtems_semaphore_ident,

		rtems_semaphore_delete, rtems_semaphore_obtain, rtems_semaphore_release, rtems_semaphore_flush.
Message	9	rtems_message_queue_create, rtems_message_queue_ident, rtems_message_queue_delete, rtems_message_queue_send, rtems_message_queue_urgent, rtems_message_queue_broadcast, rtems_message_queue_receive, rtems_message_queue_get_number_pending, rtems_message_queue_flush.
Event	2	rtems_event_send, rtems_event_receive.
Signal	2	rtems_signal_catch, rtems_signal_send.
Partition	5	rtems_partition_create, rtems_partition_ident, rtems_partition_delete, rtems_partition_get_buffer, rtems_partition_return_buffer.
Region	8	rtems_region_create, rtems_region_ident, rtems_region_delete, rtems_region_extend, rtems_region_get_segment, rtems_region_return_segment, rtems_region_get_segment_size, rtems_region_resize_segment.
Dual-Ported Memory	5	rtems_port_create, rtems_port_ident, rtems_port_delete, rtems_port_external_to_internal, rtems_port_internal_to_external.
I/O	10	rtems_io_initialize, rtems_io_register_driver, rtems_io_unregister_driver, rtems_io_register_name, rtems_io_lookup_name, rtems_io_open, rtems_io_close, rtems_io_read, rtems_io_write, rtems_io_control.
Fatal Error Manager	1	rtems_fatal_error_occurred.
Rate Monotonic	10	rtems_rate_monotonic_create, rtems_rate_monotonic_reset_all_statistics, rtems_rate_monotonic_cancel, rtems_rate_monotonic_delete, rtems_rate_monotonic_period, rtems_rate_monotonic_get_status, rtems_rate_monotonic_get_statistics, rtems_rate_monotonic_reset_statistics,

Approved for Public Release; Distribution Unlimited.

		rtems_rate_monotonic_ident, rtems_rate_monotonic_report_statistics.
Barrier	5	rtems_barrier_create, rtems_barrier_ident, rtems_barrier_delete, rtems_barrier_wait, rtems_barrier_release.
User Extensions	3	rtems_extension_create, rtems_extension_ident, rtems_extension_delete.
Multiprocessing	0	rtems_multiprocessing_announce

Each manager provides a well-defined set of services by using directives that take the place of system calls. For example, the partition manager has five directives: *rtems_partition_create* is used to create a new partition, *rtems_partition_ident* can get the ID of a partition, *rtems_partition_delete* is for deleting partition, *rtems_partition_get_buffer* is used to get a buffer from a partition, and *rtems_partition_return_buffer* returns buffer to a partition. All these directives support the partition manager to manage partitions. By using directives, developers are able to develop applications with better control of different tasks. Each manager also has internal functions that it uses to support implementation. These manager internal functions are not intended to be used by user code. Table 2 lists all the directives of each manager.

3.2.1.2 RTEMS SCORE

In addition to each manager, RTEMS has a set of functions called SCORE (Super Core). The SCORE provides services for all managers, and all managers interact with the SCORE via directives. As shown in Figure 9, the SCORE provides the following structures and functions: object, thread, chain, interrupt service routine (ISR), error handler, heap and protected heap, workspace, initialization, message, time, watch dog, and user extension. These structures and functions are key to the internal working of RTEMS. User code is not supposed to access SCORE functions that work as kernel functions of the RTEMS, however in RTEMS, user code can use SCORE functions. Therefore we address this security concern with our tagging scheme. In addition to these SCORE functions, SCORE also supports SCORE internal functions that are not intended to be accessed by managers or users.

This section introduces some of the important data structures in RTEMS: *Object*, which is a critical data structure in the SCORE, tasks threads.

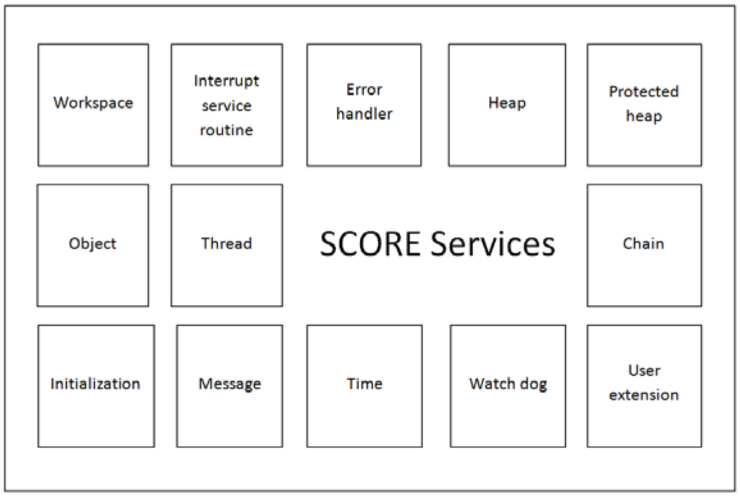


Figure 9: SCORE services

3.2.1.2.1 Object

Object is a central data structure in the SCORE that is used to interact with all user applications. Objects are created by the managers for users or managers, and can be classified into different classes depending on which manager the objects are associated with. Objects consist of six major structures: *_Object_Information*, *local_table*, *name_table*, *global_table*, *object memory*, and inactive chain. Each class of object has one *_Object_Information* structure that is used as the root structure to store pointers to the other five main structures. The *local_table* holds the pointers to allocated objects. Given an ID of an object, the object can be found by looking in the index in *local_table*. The *name_table* stores the names of the objects, so that the kernel routines can search *the name_table* for a given name and return the associated ID or returns the name of the object for a given ID. If multiprocessing is enabled, then *global_table* is also used. The inactive chain is a doubly linked list that stores the unallocated or free objects. When allocating an object, the first node of the chain will be allocated. Thus, the time looking for a free object is saved by using the inactive chain.

In addition to the *_Object_Information* structure, which is for each class of object, there is an *Objects_Information_table* structure in RTEMS. This is a global table of pointers that point to information structures for each class of object.

Currently, there are ten managers that use objects by calling functions associated with them, such as *_Objects_Allocate*, *_Objects_Free*, *_Objects_get*, etc. Each manager has one corresponding inactive chain. These functions are called with parameters that indicate the calling managers. Thus, the system knows which inactive chain to access.

3.2.1.2.2 Thread and Task

In RTEMS, a task is the smallest single thread of execution that concurrently competes for the system's resources with other [Rtems]. Each task has a task control block (TCB) associated with it. The TCB is a data structure that contains information relevant to the execution of the task. It contains a task's name, task's ID, current and starting states of the task, current priority, and so on. When creating a task, RTEMS allocates a TCB for the task. The TCB can only be freed when the task is deleted.

A task is in one of five states: executing, ready, blocked, dormant, and non-existent. RTEMS supports up to 255 levels of priority, with the lower numbers indicating higher priority. The execution mode of a task is a combination of pre-emption, asynchronous signal routine (ASR) processing, timeslicing, and interrupt level.

3.2.1.3 Initialization of RTEMS

The initialization of RTEMS starts with initializing internal RTEMS variables using the directive *rtems_initialize_data_structures*. This directive is called right after the completion of basic initialization of the board support package (BSP). The BSP is a set of facilities that help RTEMS and applications to interface with a specific hardware platform. These facilities include initialization of hardware, device drivers, user extensions, and multiprocessor communication interface (MPCI), all of which allow multiprocessing in RTEMS [Rtems].

The directive *rtems_initialize_data_structures* does the following things:

- **Initialize CPU and system state.** It specifies whether the system will support multiprocessing.
- **Initialize debug manager.**
- **Initialize API extensions.**
- **Initialize workspace.** In this case, memory can be allocated for use by RTEMS.
- **Initialize user extensions handler.**
- **Initialize ISR handler and thread handler.**
- **Initialize RTEMS APIs.** RTEMS API's include: task manager, timer manager, signal manager, event manager, message queue manager, semaphore manager, partition manager, region manager, dual ported memory manager, rate monotonic manager, and barrier manager.
- **Initialize extension manager.** Initializes all data structures related to extension manager.
- **Initialize I/O manager.** A driver table is allocated when initializing the I/O manager; thus, runtime drivers can be registered into the driver table.
- **Initialize DRV manager,** POSIX or ITRON APIs. RTEMS will initialize these API's, if the system requires these structures.
- **Initialize and start the idle task.** From here, threads can be created and scheduled.

After calling the directive *rtems_initialize_data_structures*, RTEMS uses the directives *rtems_initialize_before_drivers* and *rtems_initialize_device_drivers* to finish the initialization of device drivers. Then, user initialization tasks are created and started.

Lastly, *rtems_initialize_start_multitasking* initiates multitasking and applications can start executing.

Approved for Public Release; Distribution Unlimited.

3.2.2 Problems in RTEMS

RTEMS is an executive that provides substantial functionality, but has no protection. As an example, look at the code in Figure 10; this is the *rtems_task_delete* directive of the RTEMS task manager. This directive allows a thread to be deleted and receives an input parameter id from the calling RTEMS code or user code. In RTEMS, every thread has a unique ID by which the corresponding thread can be located. Therefore, on line 9, the *_Thread_Get* function is called to find the thread control block of the thread. If the thread ID corresponds to a local thread, then a thread control pointer is returned and location is set to *OBJECTS_LOCAL*. If the thread ID is mapped to a remote thread, then location is set to *OBJECTS_REMOTE*[3]. Otherwise, location is set to *OBJECTS_ERROR* to indicate undefined thread. If the thread is local, (lines 11-31) the function *_Objects_Get_information_id* (line 13) gets the information associated with the thread. Then the *_Thread_Close* routine (line 26-27) is used to delete the thread. This routine frees all memory associated with the thread and also removes the thread from the local object table. Therefore, no further operations can be done for this thread. By using the *_RTEMS_task_Free* routine (line 28), the task control block is freed and appended to the inactive chain of free task control blocks. If the thread is not local (line 32-38), *RTEMS_ILLEGAL_ON_REMOTE_OBJECT* will be returned.

The rest of this section analyzes this code to highlight some of the major security problems of RTEMS.

3.2.2.1 No separation between user and system

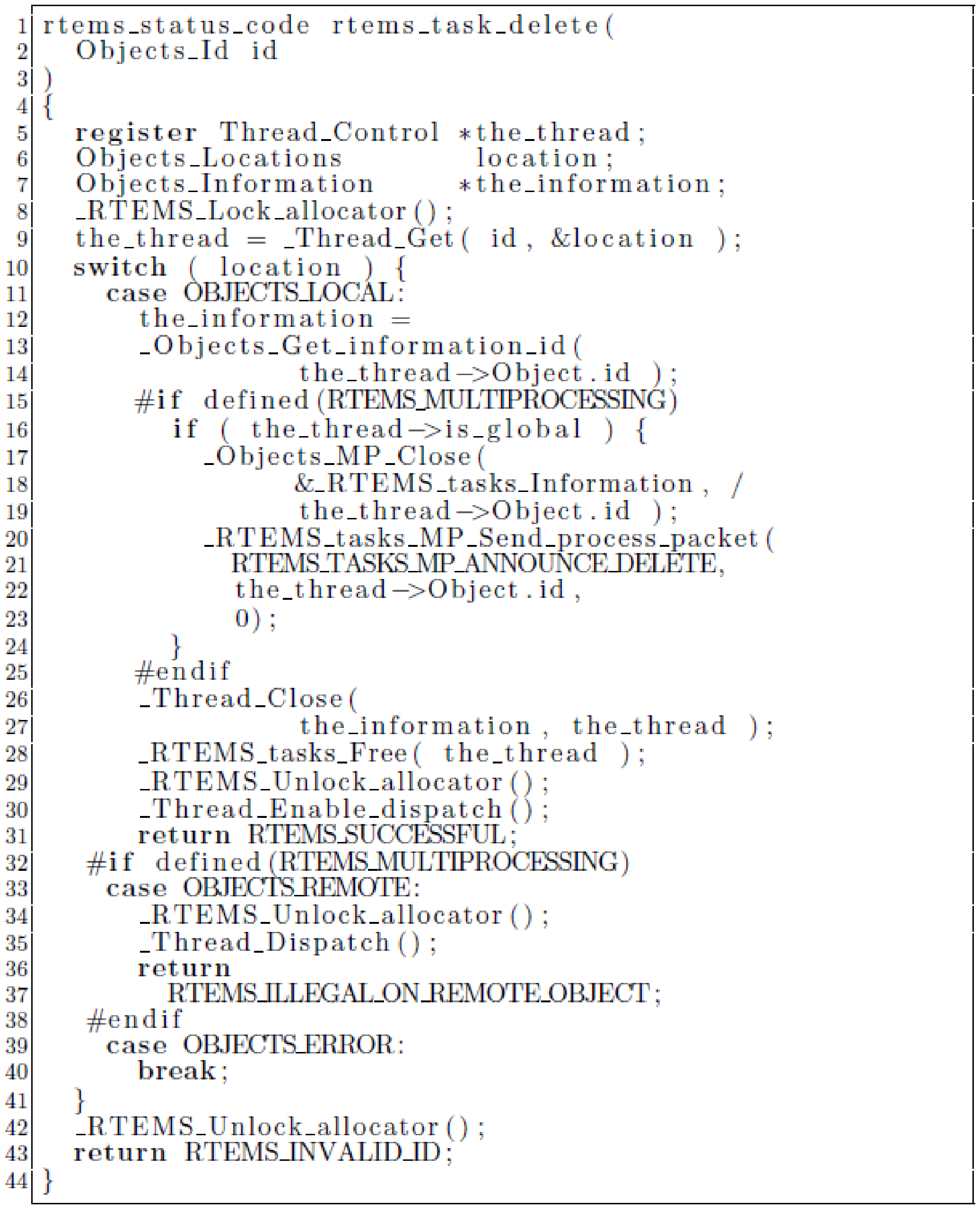
After examining the code of RTEMS and users, we found there is no separation between users and system. In traditional operating systems, the system has ultimate privileges while the user has limited privileges. In RTEMS, both the user and system have ultimate privileges. Currently user code can use all of the SCORE code, internal functions, and directives. This means that the user can do everything that the RTEMS system can. For example, the user has privileges to change the system configuration, delete system's tasks, etc. This is a design decision since RTEMS is intended for embedded systems and is built for high performance. Unfortunately, as we will see, this limits the use of RTEMS in secure environments or from safely running untrusted code.

Consider the directive code *rtems_task_delete* listed in Figure 10. There is no information about the owner of the parameter id. RTEMS checks some information about ID, such as if the thread associated with the id is a local thread, global thread or a remote thread. However, RTEMS does not check the owner of the thread. It might be an ID of a user's thread or it might be an ID of a system thread.

As critical system data must not be changed by users, user code should be separate from system code and user data should be separate from system data. Therefore, a secure RTEMS

[3] In this project, we are focusing on single processor functionality, so we will not discuss global and remote objects in detail.

needs a mechanism to distinguish user data and system data first, and then to restrict users to the minimal privileges. This mechanism can be implemented in software using security fields in the data structures, or multiple separate data structures. This separation of system data and code and user data and code can be done by using a hardware-based tagging mechanism that uses less system resources.



```
rtems_status_code rtems_task_delete(
  Objects_Id id
)
{
  register Thread_Control *the_thread;
  Objects_Locations        location;
  Objects_Information     *the_information;
  _RTEMS_Lock_allocator();
  the_thread = _Thread_Get( id, &location );
  switch ( location ) {
    case OBJECTS_LOCAL:
      the_information =
      _Objects_Get_information_id(
                the_thread->Object.id );
      #if defined(RTEMS_MULTIPROCESSING)
        if ( the_thread->is_global ) {
          _Objects_MP_Close(
                &_RTEMS_tasks_Information, /
                the_thread->Object.id );
          _RTEMS_tasks_MP_Send_process_packet(
            RTEMS_TASKS_MP_ANNOUNCE_DELETE,
            the_thread->Object.id,
            0);
        }
      #endif
      _Thread_Close(
                the_information, the_thread );
      _RTEMS_tasks_Free( the_thread );
      _RTEMS_Unlock_allocator();
      _Thread_Enable_dispatch();
      return RTEMS_SUCCESSFUL;
   #if defined(RTEMS_MULTIPROCESSING)
    case OBJECTS_REMOTE:
      _RTEMS_Unlock_allocator();
      _Thread_Dispatch();
      return
        RTEMS_ILLEGAL_ON_REMOTE_OBJECT;
   #endif
    case OBJECTS_ERROR:
      break;
  }
  _RTEMS_Unlock_allocator();
  return RTEMS_INVALID_ID;
}
```

Figure 10: Directive code for rtems_task_delete

3.2.2.2 System has no protection from users

Traditional operating systems use access control to ensure that accesses to objects are allowed. To read an object, subjects must have read permission granted. An object can be written by subjects who have write permission to the specific object. By performing access control, traditional operating systems regulate read, write, or other permissions to data and programs. Therefore, the possibilities of accessing and changing secret or critical data by unauthorized users are reduced. Depending on the operating system, access controls exist at many levels. Memory management is used to separate user objects from other users and to separate operating system objects from users. Most operating systems provide separate controls for separation of processes and some provide security mechanisms to allow processes to restrict access to specific objects.

However, in RTEMS, the system has no protection from users. Take the *rtems_task_delete* directive in Figure 10 as an example. There is no check of the identity of caller of this directive. This means that other system managers can use this directive, other system code can use it, and even users can delete a task by calling this directive. What is more, there is no restriction on who can delete which task.

Then there comes an additional security concern. RTEMS has a task directive called *rtems_task_ident* that returns the system ID associated with the thread name that is passed as a parameter to the directive. Since the user can use the *rtems_task_ident* directive to get a system ID, and there is no rule to specify who is allowed or not allowed to call the *rtems_task_delete* directive and no check on the owner of the task, users can delete system threads. In that case, malicious users can delete other users' tasks and system threads.

As RTEMS has other directives and internal functions that can delete, create, or change user data and system data, it is very dangerous to let users have permissions to use these important system codes. The example code shows a possibility that a user can delete a system thread. Therefore, in order to protect system data from users, it is better to implement a mechanism to limit the capabilities of users.

3.2.2.3 No separation of RTEMS code

There is no protection among RTEMS managers, directives, and SCORE. In RTEMS, SCORE works as the micro kernel of the system. If the SCORE code is misused by a user or attacked by a malicious user, critical security problems may occur. So it is important that SCORE code be separated from other system code and user code. Mechanisms can be implemented in RTEMS to restrict the unnecessary usage of SCORE code by others.

RTEMS has 18 managers and each manager has its own functionality. Each manager has specific directives that support the manager. The example code in Figure 10 is one of the directives of the task manager. In addition to the *rtems_task_delete* directive, the task manager has other directives such as *rtems_task_create*, *rtems_task_ident*, and *rtems_task_start*. The partition manager is used to dynamically allocate fixed size memory units. It has five directives: *rtems_partition_create* and *rtems_partition_delete* dealing with creating and deleting partitions; *rtems_partition_get_buffer* obtains a buffer from a buffer partition indicated by the partition ID parameter by returning a pointer pointed to the buffer; *rtems_partition_ident* is for getting the ID of a partition, and *rtems_partition_return_buffer* is used to return a buffer pointed to by a pointer to the partition specified by the partition ID. A manager in RTEMS can use not only the directives belonging to itself, but other managers' directives and internal functions as well. Therefore the task manager can use the directives of the partition manager. However, the five directives belonging to partition manager should be only used by managers who need to use them. The task manager should use *rtems_task_delete* to delete a task, but should not be allowed to use the *rtems_partition_delete* to delete a partition since the task manager does not use partitions. Therefore, the directives should be associated with the managers who need to use them but not the other managers. The current version of RTEMS may cause security problems if any RTEMS code is modified by an attacker because the attacked code can use any other RTEMS code that the attacker wants.

RTEMS has internal functions that help RTEMS to function correctly, inline routines that speed up the running of RTEMS, and library functions supporting some of the functionalities. These functions are used by one or more directives. Take the functions in the code of Figure 10 as examples, *_Thread_Close*, which is a SCORE function (line 26), is used only by *rtems_task_delete* directive. Whereas, *_Thread_Get* function (line9) is a SCORE function used by many other directives such as *rtems_task_variable_get*, *rtems_task_set_priority*, *rtems_task_start*, *rtems_task_suspend*, but all these directives are implemented by the task manager subsystem. Further, *_RTEMS_tasks_Free* function is an inline routine of the task manager, which is used only by the directives of the task manager. This type of internal function is called a manager internal function since it is used by code within a single manager. Likewise, the SCORE function *_Thread_Enable_dispatch* is shared by many managers, such as partition manager, message queue manager, rate monotonic manager, semaphore manager, and so on. This violates the least privilege principal and potentially leads to security problems. In a secure design, all of these functions would be manager internal functions. The detailed classification of functions are discussed in Section 3.3.3.

Given these security concerns, RTEMS code should be separated, following the principal of least privilege [Saltzer75]. In that case, we can limit the unnecessary usage of other code and eliminate more security concerns.

3.2.2.4 No separation of different users

Although RTEMS has a single user multi-threaded model of execution, it could be expanded to be a multi-user system in the future. If the system is going to be extended to a multi-user system, the system must have a method to identify the owner of user data and keep it separated from other users' data. However, RTEMS cannot tell which user is the owner of specific user data. As the example code in Figure 10 shows, the parameter passed to the *rtems_task_delete* directive (line 2) indicates the ID of a thread that is going to be deleted. There is no information about who owns the thread. If RTEMS is a multi-user system, there is no way to tell who the owner of a specific thread is. It might be a thread generated by user 1, it might be a thread generated by user 2, or it might be a system thread. Since we do not know the owner of the thread, we cannot specify who has the permission to delete it. Currently RTEMS is a single user system that contains no rules to specify who can do what. This becomes a problem when RTEMS is extended to allow more than one user.

Approved for Public Release; Distribution Unlimited.

3.3 NEW SECUITY TAGGING SCHEME

3.3.1 Tagging scheme overview

This section offers an overall description of our tagging scheme, such as the goal of using tags in RTEMS, tag format, and meanings of each field of the tag.

3.3.1.1 The goal of using tags

The purpose of a ZKOS is to move away from the concepts of a single shared kernel and into a model of integrated services for each application. The majority of operating system services are now implemented as protected library functions. The data structures and resources used by the library functions are segregated per application/user as much as possible. This model enhances both security and performance as operating system services are now treated as part of the application code space, with some important distinctions. The code for these services is protected by hardware tags, which ensures separation and access control for the functions and the resources they manage. To enhance the separation, we use tags to indicate the context of execution of this code (which user or service is making the call and therefore owns the resources used by the service), and that context defines the security classification of the resources the code can access during that particular call.

Based on the analysis of the code of RTEMS in the previous section and the goals of ZKOS, we propose a security tagging scheme for RTEMS. The main purposes of using tags in RTEMS are:

Separate system and user data and code. In RTEMS, there is no way to specify which are user's data and code and which are system's data and code. Because of this flaw, we saw a need for a mechanism to identify the owner of code and data in RTEMS. By using tags to specify the ownership of the tagged data or code, the system can securely manage the data or code and can help with the protection of system code and data from user.

Classify RTEMS code. All RTEMS system code has ultimate privileges. Since there is no need for some RTEMS code to have ultimate privileges, it is better to classify RTEMS code and only give the least privileges it needed. Traditional operating systems give the kernel code ultimate privileges. However, some parts of the kernel system do not need to have ultimate privileges. If the malicious code gets the permission to run as kernel code, it can gain ultimate privileges and do whatever it wants. By using tags, we can specify the privilege of each tagged data or code to be the least privilege it needs. Additionally, the classification of RTEMS code helps control information flow and access to code and data.

Limit the functions called by user. Since the directives of RTEMS provide a collection of services that are sufficient for users' tasks, the calling of other important kernel functions should be limited. Restriction of using these critical functions helps protect the system code and data. Supported by security tagged architecture, the access control of the functions calls provided by tags can be done at the hardware level which reduces the consumption of software resources.

Protect the return values. To prevent users from changing the return values (such as task id) and sending back modified values to system code, some return values to the user should be protected. Combined with other security mechanisms, this helps prevent malicious attacks.

Control the information flows in RTEMS. To protect the RTEMS data from leaking information to unauthorized users, information flows in RTEMS need to be controlled properly. Tags carry information about the tagged data, which help track and control the information flow.

3.3.1.2 Overview of tag format and meanings

Based on the goals of using tags, in our tagging scheme, a tag consists of three fields: owner, code-space and control-bits. The owner field helps separate system and user resources (i.e., code and data). It identifies the owner of the data and code. The second field of the tag specifies which code space created the data. Together the owner and code space specify the security class of the tag. The third field of the tag, control-bits, is used for further refinement of controls. It consists of one copy bit and three memory type bits and four reserved bits. The details of the tag format are discussed in the next section.

3.3.2 New Security Tagging Scheme Design

As discussed in the evaluation of the source code of RTEMS in Section 3.2, we found that RTEMS has many security problems that need to be solved if it is to be used in a multi-user, secure environment. In Section 3.3, we propose a three field tag consisting of the owner field, code-space field, and control-bits field. Each of these three fields helps maintain the correctness and security of RTEMS. Although our new tagging scheme is focused on the security problems of RTEMS, the problems are common to many types of operating systems and thus the tagging scheme should be usable for other operating systems as well. This tagging scheme will allow us to use the principle of least privilege in protecting resources in the system with a finer-granularity than traditional supervisor and user modes.

In addition to using tags to control data access, the idea of access control is implemented in our tagging scheme to control the execution of functions. To prevent malicious use of critical functions, we provide rules specifying who can call which functions. In addition to the access control mechanism, a lattice model of information flow control is used to control the information flows within RTEMS. In Section 3.3, we propose our tagging rules for C language programs. In Section 3.4, we refine the rules for an assembly language implementation on a SPARC processor.

Since this project will implement tagging on the SPARC processor, the important features of SPARC, such as register windows and special purpose registers, have to be considered. A study of SPARC instructions revealed that the information control rules designed for the C programming language needed to be refined to fit into the SPARC architecture. Some of the rules for SPARC instructions are similar to the rules for the C language, but some of the security features require the implementation of new instructions. What is more, at the assembly language level there are more things that need to be considered, such as the runtime stack, use of registers, and the program counter (PC). As a result, the remaining part of this chapter provides an assembly language level implementation of the tagging scheme. Rules are provided for assembly language instructions that have direct C language counter parts.

The remainder of Section 3.3 is organized as follows: The design of our tag format is illustrated in Section 3.3.1. In Section 3.3.2, we introduce the tagging scheme by illustrating how

Approved for Public Release; Distribution Unlimited.

it solves the security problems we found. Security tagging rules are presented in Section 3.3.3, and the tagging rules for C-language are in Section 3.3.4. Section 3.3.5 discusses the tag manager, which is used to manage the tagging scheme in RTEMS. By using a sample of the RTEMS code, we show how the security tagging scheme works for RTEMS in Section 3.3.6.

3.3.3 Tag format and meanings of each field

As shown in Figure 11, the tag in our tagging scheme consists of three fields: Owner, Code-space, and Control-bits. The tag can be written as (<Owner>, <Code-space>, <Control-bits>).

The Owner field and Code-space field are used to provide a measure of modularity to the operating system. Secure systems require a design that uses the concepts of least privilege, where data and code are designed in modular units controlling only limited resources of the operating system. We use the Owner field to indicate the entity that owns the resource managed by the code module. The Code-space field indicates the code modules that are currently executing and/or the code modules that are authorized to access specific operating system resources (e.g. data structures and OS functions), in support of least privilege.

For example, the task manager code space is authorized to manage tasks of the system, but tasks may be created on behalf of different users or managers. Therefore we use the owner field to further refine privileges so that the task manager will have hardware supported access control that ensures that the task manager manages tasks only for users who own those tasks (e.g. An user cannot request the deletion of another user's task.)

The Control-bits field is used to even further support least privilege by providing some typing and access control information to the system resources. We intend to extend this field with some of the attack prevention technologies of Suh and others. The bit representation in Figure 11 is a current recommendation. The layout for these bits is described in Section 3.4.4

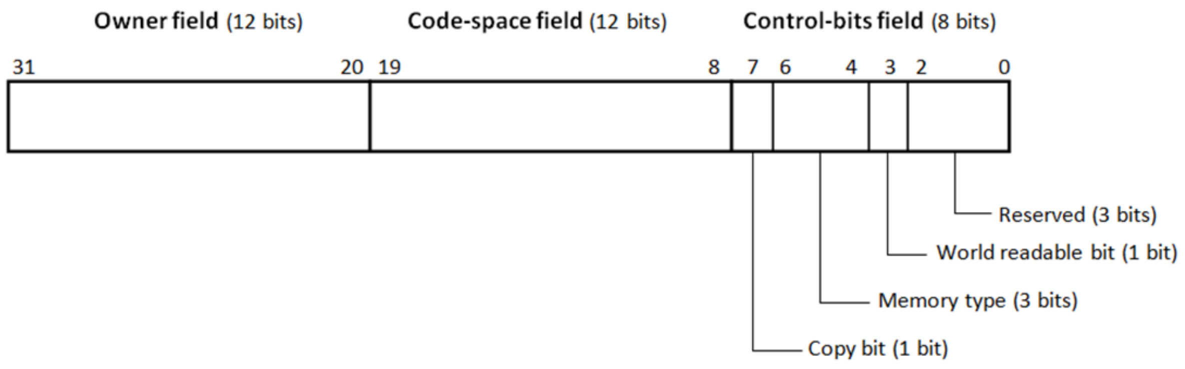


Figure 11: Tag format

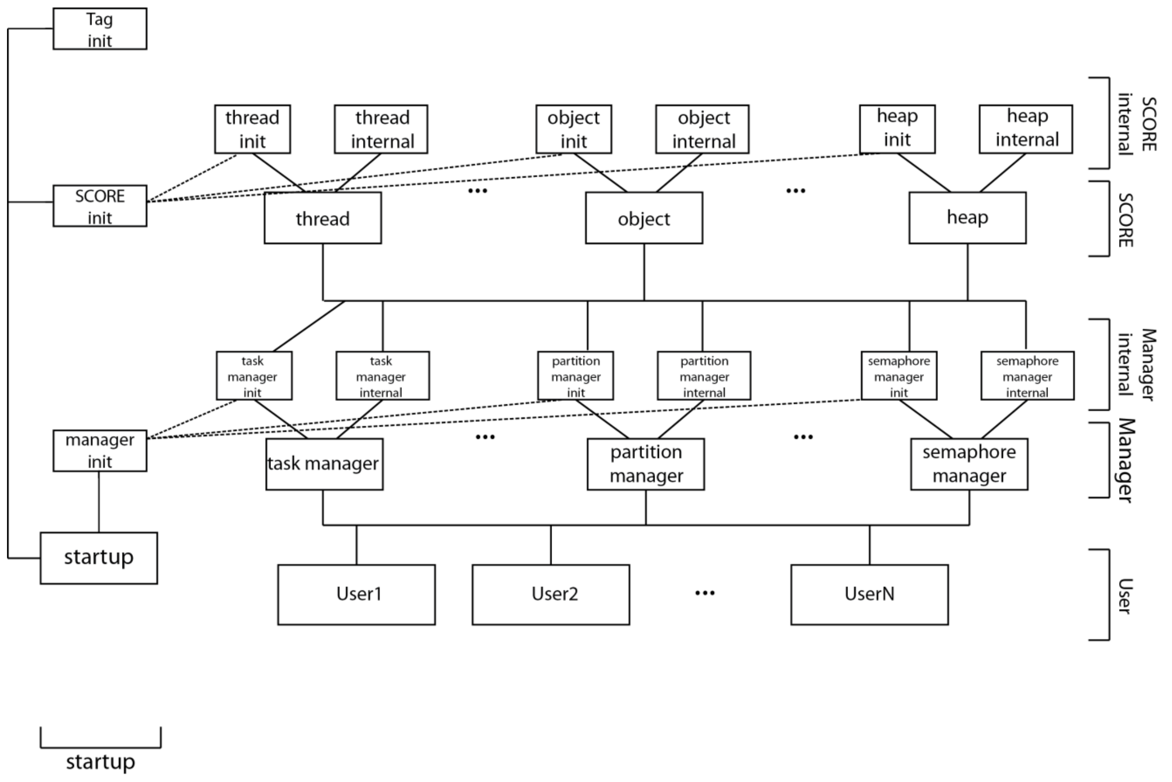


Figure 12: Classification of RTEMS code and users

3.3.3.1 Owner field

In our tagging scheme, the first field, Owner field, helps separate system code/data and user code/data. It indicates who is the owner of the data or code. As shown in the Manager group of Table 3, the values of the Owner field can be classified into six major classes: *SCORE internal, SCORE, Manager internal, Manager, Startup* and *User*. We further divided the *SCORE internal class* into *SCORE internal init* group and *SCORE internal* group. The *Manager internal* class is also broken into two groups: *Manager internal init* and *Manager internal*. The possible values of each class are listed in Table 3. By using the owner field, the owner of data or code can be easily identified. For example, the first field in the tag (object, <Code-space>, <Control-bits>) indicates that the data or code associated with this tag is object code or object data; object is one of the SCORE functions. A tag (task manager, <Code-space>, <Control-bits>) shows that the data or code is owned by the task manager. In the remaining sections, <SCORE> in the tag means this field could be one of the possible values in the SCORE class, and the same holds for the Startup class, SCORE internal class, Manager class, Manager internal class and User class. Although RTEMS currently only supports a single user, our plan is to expand it to a multiuser system.

Table 3: Possible values of the Owner field and Code-space field

Group	Class	Count	Possible values
SCORE	SCORE internal	13	thread internal, object internal, chain internal, initialization internal, heap internal, protected heap internal, message internal, error internal, time internal, ISR internal, workspace internal, watch dog internal, user extension internal
	SCORE init	13	thread init, object init, chain init, initialization init, heap init, protected heap init, message init, error init, time init, ISR init, workspace init, watch dog init, user extension init
	SCORE	13	thread, object, chain, initialization, heap, protected heap, message, error, time, ISR, workspace, watch dog, user extension
Manager	Manager internal	20	initialization manager internal, task manager internal, interrupt manager internal, clock manager internal, timer manager internal, semaphore manager internal, message manager internal, event manager internal, signal manager internal, partition manager internal, region manager internal, dual ported memory manager internal, I/O manager internal, fatal error manager internal, rate monotonic manager internal, user extensions manager internal, multiprocessing manager internal, barrier manager internal, tag manager internal, user manager internal
	Manager init	20	initialization manager init, task manager init, interrupt manager init, clock manager init, timer manager init, semaphore manager init, message manager init, event manager init, signal manager init, partition manager init, region manager init, dual ported memory manager init, I/O manager init, fatal error manager init, rate monotonic manager init, user extensions manager init, multiprocessing manager init, barrier manager init, tag manager init, user manager init
	Manager	20	initialization manager, task manager, interrupt manager, clock manager, timer manager, semaphore manager, message manager, event manager, signal manager, partition manager, region manager, dual ported memory manager, I/O manager, fatal error manager, rate monotonic manager, user extensions manager, multiprocessing manager, barrier manager, tag manager, user manager
Startup	Startup	4	Startup, manager init, SCORE init, Tag init
User	User	N	user1, user2, ..., userN

Therefore, multiple users[4] are listed in the possible values of the User class. Having different users for the Owner field ensures that a user can only access its own data and code; but not other users' resources. The bit representation field can be found in Section 3.4.4.

3.3.3.2 Code-space field

The second field of the tag is Code-space. This field shows which code space should manage the data or code. In addition to this, the Code-space field is also critical for information flow control and memory access control. The possible values of the Code-space field are the same as the Owner field (Table 3). The Code-space can be <User>, <Manager>, <Manager internal>, <SCORE>, <SCORE internal> and <Startup>. For example, the tag (User1, Manager1[5], <Control-bits>) means the data is created in the manager1's code for user1.

Code-space is used to show the class of the code or data and helps control function calls. We do not want users to use some of the system functions, such as SCORE, SCORE internal, and Manager internal functions. Therefore, firstly, we use the Code-space to indicate which class the code belongs to (Section 3.3.4.2), and then provide rules to control which classes of code can be used which other classes of code (Section 3.3.4.3). The bit representation for the Code-space field can be found in Section 3.4.4.1.

3.3.3.3 Control-bits field

The Control-bits field is used for further control. We started with single copy bit which indicates whether a return value has been modified. The highest bit (bit 7), copy bit, allows user code to have a copy of a trusted data value (i.e., a task ID) as long as it is not changed. The notation $\overline{cp}$ means the copy bit is not set and cp indicates the copy bit is set. The return value to a user will be tagged with the security class of the directive and will have the copy bit set. If the copy bit remains set, it means that user has not made any change to the value. If the copy bit is not set, the data is treated as modified data and will not be accepted when used as a parameter to a directive. For example, if the user1 uses directive code from task manager to get a tag identifier, the directive returns an identifier tagged (user1, task manager, cp). If the user modifies this returned ID, the tag will be changed to (user1, user1, $\overline{cp}$) to indicates this id has been changed (Section 3.3.5.4).

We allocate three bits (bits 6 to 4) for memory type. To further protect the memory, we divided memory into to three classes: stack memory, code memory, and data memory. These three bits specify the memory type, such as readable, writable, read-only, executable, entry point, data memory, and stack memory (Section 3.4.4.2).

We use a world-readable bit (bit 3) to indicate that the tagged data can be read by all entities; used when the system (higher level) wants to give the user (lower level) permission to access some data, such as configuration data. We reserve the remaining 3 bits (bit 2 to 0) for future use.

[4] For the User class, each user could be further divided into tasks owned by the user, such as user1-task1, user1-task2, user2-task7. However, this is beyond the scope of this project.

[5] We use manager1 as a place holder for a specific manager name, such as task manager, partition manager, etc.

3.3.4 Tagging features needed in RTEMS

In the Section 3.2, security problems of RTEMS were discussed; basically RTEMS has no protection. The user has ultimate privileges, the same as the operating system. Given data, there is no way to tell if the data is from the user or from the system. Users can use any system code and can access any system data. Because there is no separation between users, if there is more than one user then every user can access other user's data and even make changes to the data or delete it. In the previous section (Section 3.3.3) we introduced our tag format and meanings of each field. This section discusses how to use each field of the tag to enhance RTEMS security.

3.3.4.1 Separation of system and user

To separate user code and system code, we add one field in the tag to show the owner of the code. Similarly, the owner field also shows the owner of data.

When initializing the RTEMS system, system code is tagged properly by the tag manager (Section 3.3.7). The system code includes Startup, SCORE, SCORE internal, Manager and Manager internal. The SCORE code and data have tags to show that the code is owned by SCORE. The manager code and data have tags to indicate that the code and data belong to managers. The data and code of SCORE internal functions and manager internal functions are tagged as SCORE internal and Manager internal in the owner field of the tag. User1's code will be tagged as user1 and all the data generated by the user1 or generated for the user will have user1 tags as well.

Adding the owner field in the tag is important for protection of the system. Because the ownership of the code and data are known, we can add other mechanisms to limit the capabilities of the user code and data.

3.3.4.2 Classifying RTEMS code

The second field in tags is the Code-space field. This field allows us to have a modular design to RTEMS, indicating which code space created the data and therefore enforce least privilege. The first field and the second field of the tag are combined to show the security class of the tag. These classes form a partial ordering of tags, which we discuss further in Section 3.3.6.

For example, a variable declared in user code is tagged (user1, user1)[6] which shows the security class of the data, because it is a variable for user1 and the variable is created by user1 code. However, a variable tagged as (user1, task manager) indicates that it is a variable for user1 that is created by the task manager's code. A variable for user1, used in partition internal functions, is tagged (user1, partition internal).

[6] The Control-bits field of the tag is not part of the security class, so we do not show it here.

```
 1 /* User code: */
 2 user()
 3 {
 4     ...
 5     int x = 0;     \\ tag(x) = (user1, user1)
 6     int y = 0;     \\ tag(y) = (user1, user1)
 7     y = directive( x );
 8     ...
 9 }
10
11 /* Manager3 code: */
12 \\ tag(directive function) = (manager3, manager3)
13 directive(int t)  \\ tag(t) = (user1, user1)
14 {
15     int p = 0;     \\ tag(p) = (user1, manager3)
16     p = t + 2;
17     return p;
18 }
```

Figure 13: Sample C code 1

The sample code in Figure 13 shows user1 making a call to *directive(x)* from a manager we call manager3. It illustrates the idea of the Code-space field and Owner field of the tag. In the code, lines 3 to 7 show the user code and lines 11 to 15 show a directive.

Variables x and y are declared in user1 code on lines 5 and 6, so both of these variables are tagged (user1, user1).

The directive function has an initial tag (manager3, manager3). This means that this directive function is one of the directives of manager3.

On line 7, the user code passes *x* as a parameter to the directive function.

In the directive function, parameter t (line 13) gets the value of *x* which is tagged (user1, user1), so t also gets tag (user1, user1).

Variable *p* is declared locally in directive code, which is called by the user, so *p* is tagged as (user1, manager3).

The tag of p, (user1, manager3), indicates that *p* is generated in manager3 for user1, during a call from the user1. This way when other users call the same directive function, objects managed by the directive will be tagged differently for each user.

At this time, we are leaving the code space more generic to simplify the experimental implementation being developed at Cornell University and the simulator we are developing.

3.3.4.3 Protection of Calls and Functions

To prevent a user from misusing or attacking RTEMS code, we implement access control mechanisms for the functions and calls.

Conventionally, access control mechanisms specify who can access system resources. Basically, the protected entities are called objects and the active objects are called subjects. A specific subject may have different access permissions to different objects. The permissions can be read, write, execute, etc. For example, subject1 could be permitted to read the contents of object1, but not write to it and execute it. Subject1 could execute object2, and read but not write to it. In this way, the system has rules for all subjects and objects. When a subject accesses an object, the system checks whether the subject has the right to access the particular object. If the subject is not allowed to access the object, system will deny the access.

In our approach, all system functions and calls have tags associated with them.

- Startup functions have tags (<start>, <start>).
- SCORE internal functions have tags (<SCORE internal>, <SCORE internal>).
- SCORE internal init functions have tags (<SCORE internal init>, <SCORE internal init>).
- SCORE functions have tags (<SCORE>, <SCORE>).
- Manager internal functions have tags (<Manager internal>, <Manager internal>).
- Manager internal init functions have tags (<Manager internal init>, <Manager internal init>).
- Manager functions (directives) are tagged (<manager>, <manager>).
- User functions are tagged (<user>, <user>).

We use the idea of access control in our tagging scheme. When executing a function call, the tag manager will first check the access control rules for function calls to see if the code has permission to call the function. We call this “function execution control”.

By performing access control for the function calls, we are able to control who can execute which function. For example, we only want user1 code, which is created in user1, to call its own functions and managers’ code (directives) but not SCORE code and other functions. System code in SCORE is allowed to call SCORE internal functions. System code in Manager is allowed to call Manager internal functions. Both Manager code and Manager internal code are allowed to use SCORE functions. All the system functions (Startup, SCORE, SCORE internal, Manager, Manager internal) are not allowed to call User functions. Note we divided Manager internal functions into Manager internal functions and Manager internal init functions. We only allow Managers to call their corresponding Manager internal functions but they are not allowed to call other managers' internal functions and any of the Manager internal init functions. SCORE functions are similar to the Manager functions. Because the first two fields are sufficient to control the access, we only show the Owner field and Code-space field of the tag when giving the rules. Table 4 shows the specific function execution control rules.

Table 4: Function Call Tag Propagation Rules

Caller's Tag	Callee's Tag	Result Tag
(<user1>, <User1-space>)	(<user1>, <user1-space>)	(<user1>, <user1-space>)
	(<user2>, <user2-space>)	Not allowed
	(<manager1>, <manager1-space>)	(<user1>,<manager1-space>)
	(<manager1 internal>, <manager1 internal-space>)	Not allowed
	(<SCORE1>, <SCORE1-space>)	Not allowed
	(<SCORE1 internal>, <SCORE1 internal-space>)	Not allowed
	(<Startup>, <Startup-space>)	Not allowed
(<user1>, <manager1-space>)	(<manager1>, <manager1-space>)	(<user1>, <manager1-space>)
	(manager1 internal inline, manager1 internal inline-space)	(<user1>, manager1 internal inline-space)
	Others	Not allowed
(<user1>, <manager1 internal inline-space>)	(manager1 internal inline, manager1 internal inline-space)	(<user1>, <manager1 internal inline-space>)
	(<SCORE1>, <SCORE1-space>)	(<user1>, <SCORE1-space>)
	Others	Not allowed
(<user1>, <SCORE1-space>)	(<SCORE1>, <SCORE1-space>)	(<user1>, <SCORE1-space>)
	(SCORE1 internal inline, SCORE1 internal inline-space)	(<user1>, SCORE1 internal inline-space)
	Others	Not allowed
(startup, startup-space)	(startup, startup-space)	(startup, startup-space)
	(Tag init, Tag init-space)	(Startup, Tag init-space)
	(SCORE init, SCORE init-space)	(Startup, SCORE init-space)
	(manager init, manager init-space)	(Startup, manager init-space)
	Others	Not allowed
(Tag init, Tag init-space)	(Tag init, Tag init-space)	(Tag init, Tag init-space)
	Others	Not allowed
(SCORE init, SCORE init-space)	(SCORE init, SCORE init-space)	(SCORE init, SCORE init-space)
	(<SCORE1 internal init>, <SCORE1 internal init-space>)	(<SCORE1 init>, <SCORE1 internal init-space>)
	Others	Not allowed
(manager init, manager init-space)	(manager init, manager init-space)	(manager init, manager init-space)
	(<manager1 internal init>, <manager1 internal init-space>)	(<manager1 init>, <manager1 internal init-space>)
	Others	Not allowed

In Table 4 the first column specifies the identity of the calling subject. Here we have classified the subject based on the tag of the current executing thread. The second column specifies the function called, based on the tag of the code. The last column indicates the tag of the thread when it begins execution of the code. In the table, the value <manager1> represents one of the possible values of *Manager* class. The tag (<manager1>, <manager1>) indicates that one of the possible managers and its corresponding code-space, such as semaphore manager and semaphore manager. We use manager1 and manager2 to indicate different managers. In our rules, a thread tagged (<user1>, <manager1 internal>) can call a function that is tagged (manager1 internal, manager1 internal). Note the tag (manager1 internal, manager1 internal) means the specific manager's internal function but not any other managers' internal functions. For example, region directives from region manager can only call region manager's internal functions; they can't call other managers' internal functions.

The function execution control rules can be changed to be more specific. For example, we can set up a rule that user1 data, which is created in user1 code-space (tagged (user1, user1)), is not allowed to call an interrupt directive that has tag (interrupt manager, interrupt manager) or region directive, which has tag (region manager, region manager). We can prohibit timer manager internal function from calling protected heap functions (one of the SCORE functions). By implementing this, the function execution control can be more explicit and more flexible.

3.3.4.4 Protect return values

The third field in the tag, Control-bits, is used for additional protections. At the C-code level

```
/* User code: */
user()
{
  int x = 0;           \\tag(x) = (user1, user1, c̅p̅)
  int y = 0;           \\tag(y) = (user1, user1, c̅p̅)
  y = directive(x);    \\tag(y) = (user1, manager1, cp)
  x = y + 4;           \\tag(x) = (user1, user1, c̅p̅)
  directive2(x);       \\tag(x) = (user1, user1, c̅p̅)
  directive2(y);       \\tag(y) = (user1, manager1, cp)
}

/* Directive code: */

\\tag(directive function) = (manager1, manager1, c̅p̅)
directive(int t)        \\ tag(t) = (user1, user1, c̅p̅)
{
  int p = 0;            \\ tag(p) = (user1, manager1, c̅p̅)
  p = t + 2;
  rtems_tag_copy(&p);   \\set the copy bit
  return p;             \\ tag(p) = (user1, manager1, cp)
}

\\ tag(directive2 function) = (manager1, manager1, c̅p̅)
directive2(int q)
{
  rtems_tag_validate_copy(q);   \\validate the copy bit
  ....
}
```

Figure 14: Sample C code 2

we have implemented one control-bit, the copy-bit, which is used to protect a value returned from a directive, so that the user may keep it and pass it back later, but not change it. When user code gets a value from a higher level code, for example when a user variable gets a return value from directives, the copy-bit in the tag of the user variable can be set. We use cp to denote copy-bit set and $\overline{cp}$ for copy-bit not set. If the copy-bit is set, it indicates that the value is from higher level code and the user has not changed it. If the variable is changed, for example by assigning a new value to the variable or adding a constant to it, the copy-bit in the tag of the variable is reset. If the copy-bit is reset, this means the value of the variable is changed and higher level code should not trust the value and use it any more.

The code in Figure 14 shows tags with the three fields, including the copy-bit. Lines 2-10 are user code, lines 15-21 are directive code which is for manager1 and lines 24-28 are directive2 code which is also for manager1. Line 19 shows a call to a new directive from our tag manager (Section 3.3.7). This directive sets the copy-bit for p before returning it to the user.

On line 6, variable y gets the return value from the directive function, so the tag of y changed from (user1, user1, $\overline{cp}$) to (user1, manager1, cp).

Then on line 7, user changes the value of x by getting the result of adding 4 to y. Therefore, the copy-bit in the tag of x will be changed to $\overline{cp}$ and tag will be (user1, user1, $\overline{cp}$). This means that the value is from user space, and not from directive space.

On lines 8 and 9, user calls directive2. The tag manager checks whether the user can call directive2. Since user code has tag (user1, user1, $\overline{cp}$) and directive2 has tag (manager1, manager1, $\overline{cp}$), according to our function execution control rules (Section 3.3.6.7) both of the calls to directive2 are allowed.

On line 8, the user passes the parameter x back to directive2, where the tag shows that the variable is from user1 and not from itself, therefore directive2 will not use it. Otherwise if the user does not modify the returned value, such as on line 9, parameter y passes to directive2 still tagged (user1, manager1, cp). Directive2 has a *rtems_tag_validate_copy* directive call (on line 26) that is used to check the validation of a tag. After directive2 code validates the tag of the parameter, it trusts the parameter and executes the directive2 code because it shows that the parameter is for the user and is from directive space. A more detailed example of the use of copy-bit validation is given in Section 3.3.8

Not all directive code needs to check the tags of parameters. Only some critical directives need to perform this check, for example, the directives which delete things for the user must check that the parameter that is passed to the directive is the correct one that was created for the particular user.

3.3.5 Information flow control

This section introduces the security lattice for our tagging scheme.

3.3.5.1 Security Lattice

Our security classes are arranged in a partial ordering (using operator ≤) that is mapped to a lattice. The lattice model of information flow deals with channels of information flow and policies. For the lattice model of information flow, there exists a lattice with a finite set of security classes and a partial ordering between the classes of security classes. The nodes of the lattice represent the individual security classes. The notation ≤ denotes the partial ordering relation between two classes of information. For example, A ≤ B means class A is at a lower or equal level than class B. In a lattice, the ≤ operator is reflexive, transitive, and antisymmetric:

- **Reflexive:** A ≤ A
- **Transitive:** if A ≤ B and B ≤ C, then A ≤ C
- **Antisymmetric:** if A ≤ B and B ≤ A, then A = B

The simplest lattice may include only two security classes: High and Low. A multilevel lattice can have more than two security classes, as seen in Figure 15, which has ten classes. The arrows in the figure are used to denote the relation of ≤. An arrow from class RED to BLUE means RED ≤ BLUE.

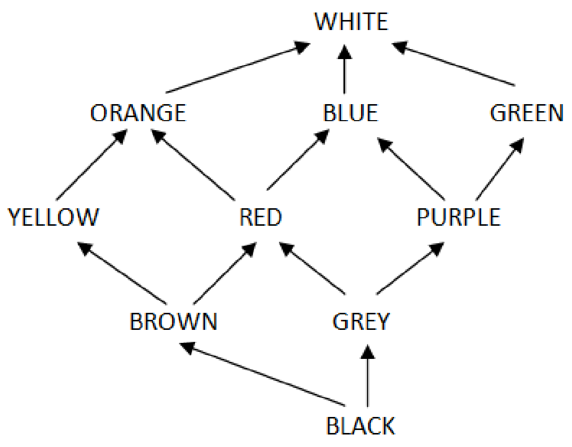


Figure 15: Multilevel lattice model

The notation ⊕ denotes the least upper bound (LUB) operation. In a lattice, there exists a class C = A ⊕ B, such that:

- If A ≤ C and B ≤ C, and A ≤ D and B ≤ D, then C ≤ D for all D in the lattice.

For example, in the lattice shown in Figure 15:

- $BROWN ⊕ PURPLE = BLUE$,
- $YELLOW ⊕ RED = ORANGE$,

- $YELLOW ⊕ PURPLE = WHITE$,

3.3.5.2 Confidentiality and Security Lattice

The classical security lattice can be used to support the confidentiality security policy of a system. Following the model of Bell and LaPadula [Bell75], we can define two simple security predicates:

Simple Security Condition (SCC or "no read-up"). A subject can only read information from an object at the same or lower security classification. Read is allowed if and only if $\underline{O} \leq \underline{S}$, where $\underline{O}$ is the security clearance of the object, and $\underline{S}$ is the security classification of the subject.

Star Property (-property or "no write-down").* A subject can only write information to an object at the same or higher security classification. Write is allowed if and only if $\underline{S} \leq \underline{O}$, where $\underline{O}$ is the security clearance of the object, and $\underline{S}$ is the security classification of the subject.

These predicates, which can be enhanced with additional discretionary restrictions (e.g. and the owner permits the action), control the copying of information from higher levels to lower levels. This will be very useful in our tagging rules as we protect user and system data from unauthorized copying. We even use these rules to prevent indirect or covert use of data in the rules for the conditional expressions in if and while statements.

In 1974, Fenton [Fen74] proposed a new abstract machine, the data mark machine (DMM). This machine used the lattice model of information flow control. Fenton's lattice was based on multilevel security confidentiality controls. The DMM was an extension of a Minsky machine [Minsky67] that adds tags for each register and memory location to show their security classes. In addition to the tags for each register and memory, DMM introduces a security class (a tag) for the program counter (PC), denoted $\underline{PC}$, which helps to control the information flow of conditional branches. The PC indicates the context of the current running thread; therefore, the tag of PC is the tag of the current executing thread's tag.

3.3.5.3 Integrity and Security Lattice

The classical security lattice can be used to support an integrity security policy of a system. Following the model of Biba [Biba77], we can define two simple security predicates:

Simple Integrity Condition (SIC or "no read-down")}. A subject can only read information from an object at the same or higher integrity classification. Read is allowed if and only if $\underline{S} \leq \underline{O}$, where $\underline{O}$ is the integrity clearance of the object, and $\underline{S}$ is the integrity classification of the subject.

Star Integrity Property (*-integrity property or "no write-up")} A subject can only write information to an object at the same or lower integrity classification. Write is allowed if and only if $\underline{O} \leq \underline{S}$, where $\underline{O}$ is the integrity clearance of the object, and $\underline{S}$ is the integrity classification of the subject.

The intent of the Biba integrity model is to ensure that the trust placed in data is only as high as its least trustworthy source. We don't want untrusted subjects, or even untrusted data, corrupting our system. This model is mathematically the dual of the Bell-LaPadula model, and if not combined correctly the two would prevent the flow of information up-and-down the lattice.

In our tagging scheme, we trust operating system code more than user code, and therefore implicitly assume a higher level of integrity and trust in the actions of this code. When storing data through an assignment, we not only perform a confidentiality check to ensure that the source expression can be written to the target variables memory, we check the integrity of the thread performing the action to ensure that the assignment does not perform an unauthorized overwrite of a value.

3.3.5.4 The Lattice and our Tagging Scheme

Since the first two fields of our tag, Owner field and Code-space field are used to define the security class of the data, we can implement ideas similar to those of the DMM in our tagging scheme to control the information flows within RTEMS. Our solution is a bit different from DMM since we are defining an operating system hierarchy with confidentiality and integrity controls.

We defined a lattice for the information flow control within RTEMS, and the Owner field and Code-space field of the tags are used to represent the security classes of the data and code. Our lattice is shown in Figure 16. Basically, we have two identical lattices, one is for the Owner field and the other is for the Code-space field. The lattice may be changed to provide more control and flexibility in the future. To simplify the diagram, we assume that boxes higher on the figure have higher integrity and security classes than those below them, and that the $\leq$, $\geq$, $>$, $<$, and $\oplus$ operators support that hierarchy.

In our model the security class of an *a* will be denoted as $\underline{a}$, and it can be written as $\underline{a}$ = {Owner(a), Code-space(a)}, where owner(a) represents the owner field of *a*'s tag and Code-space(a) denotes the Code-space field in the tag of *a*. We have ignored the control-bits in this discussion since they are used separately from the lattice-based controls and formulas. The definition of the least upper bound of the security classes of two tags, tag(a) and tag(b) is:

- $\underline{a}$ = {Owner(a), Code-space(a)}, and $\underline{b}$ = {Owner(b), Code-space(b)}, then $\underline{a} \oplus \underline{b}$ = {Owner(a) $\oplus$ Owner(b), Code-space(a) $\oplus$ Code-space(b)}.

For example, if x has a tag (user1, user1, $\overline{cp}$) associated with it and y has a tag (semaphore manager, semaphore manager, $\overline{cp}$), with the security classes $\underline{x}$ = {user1, user1} and $\underline{y}$ = {semaphore manager, semaphore manager}, then according to our lattice, $\underline{x} \oplus \underline{y}$ = {semaphore manager, semaphore manager}.

In our model, the definition of the $\geq$ is:

- If Owner(a) $\geq$ Owner(b), and Code-space(a) $\geq$ Code-space(b), then $\underline{a} \geq \underline{b}$.

3.3.6 C-Language tagging rules

In the previous section we introduced our tagging rules. Those rules require that all variables and code receive security tags. At the start of the program, and as each subroutine is called, the variable and thread tags are initialized with the correct security classification and memory types. This section discusses how we use those classifications, in the context of the C programming language, to define tagging rules needed to satisfy our security concerns based on the partial ordering and lattice concepts just introduced.

3.3.6.1 Tagging rules for basic values in C

Basic values in C follow the following tagging rules:

- During execution of a program, the tag of the current thread is denoted using the programmer counter tag PC.
- The tag of the variable *a* is the tag of the memory location of *a* and is denoted a.
- The tag of a literal, or constant, *n*, is the same as the tag of the PC, PC. We made this choice since the use of the literal is controlled by the current thread.

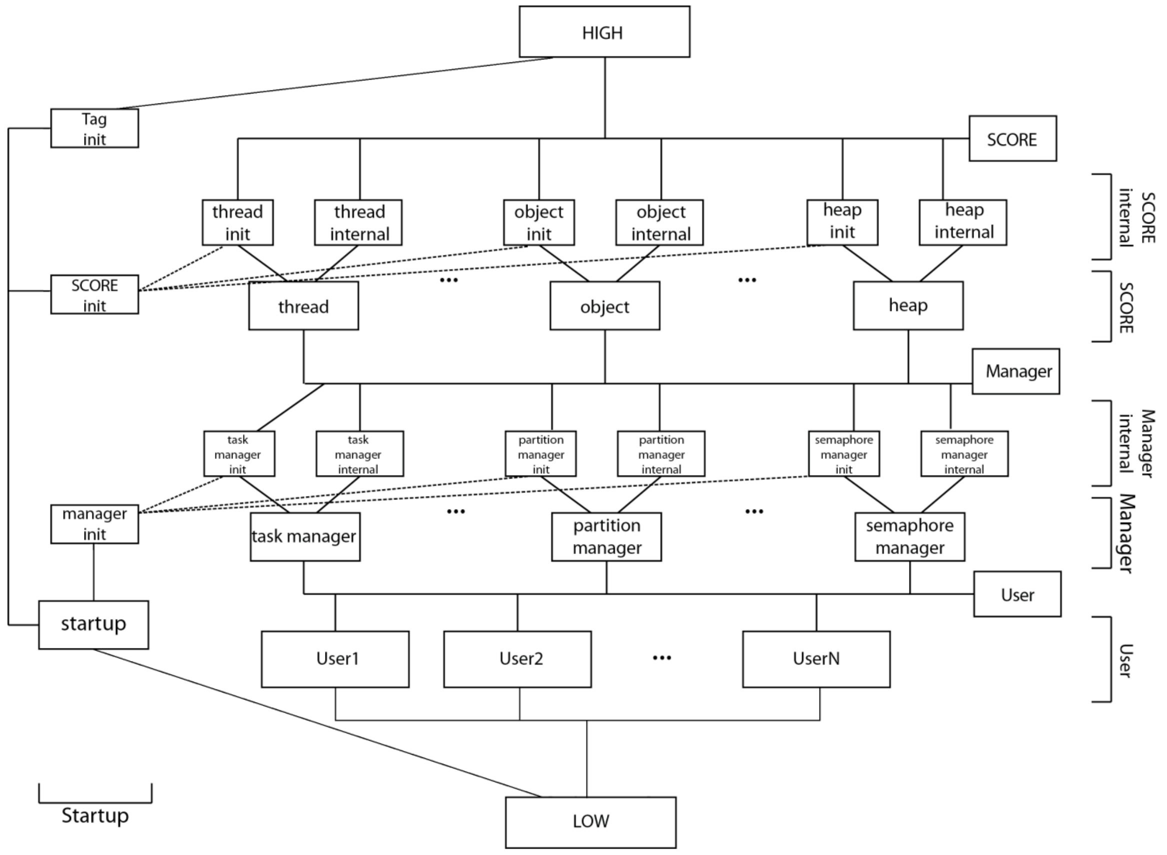


Figure 16: Our lattice designed for Owner field and Code-space field

- The tag of an array item, a[i] is the least upper bound of the tag of the index to the array and the tag of the memory location referenced by the array: a[i] = i ⊕ [a+i] where [a+i] denotes the memory address referenced by a[i].

- The tag of a value referenced by a pointer, **p* or structure p->fld or p.fld is the tag of the memory location referenced. For example, $\underline{*p} = \underline{[p]}$ where [p] denotes the memory address referenced by *p*.
- All code will be tagged as read/only, executable memory (see Section 3.4.4 for details of binary representation of tags and the memory type tagging.)
- All entry points to functions will be tagged as function entry points (to avoid problems with the use of function pointers).
- All data memory will be tagged as read-write data memory.

3.3.6.2 Rules for arithmetic and logic operations

The rules for arithmetic and logic operations are used to specify the resulting tag of the resulting value of the operation. For example, in $a + b = c$, *a* and *b* are two operands, and c is the result of this operation. The security class of the result depends on the copy-bits and the security classes of the two operands. A set copy-bit indicates that the tag of a value consists of the tag of the directive that created the value. Any change to the value ignores the directive's tag. Therefore the copy-bit in the tag of *a* is cp and the copy-bit of *b* is $\overline{cp}$, then we just need to consider the security class of *b*. If the copy-bit of both operands are $\overline{cp}$, then the security classes of both *a* and *b* need to be used and the security class of the result will be $\underline{a} \oplus \underline{b}$. If *a* or *b* are constants, they have the tag of the PC. In the tag of the result of arithmetic and logic operations, the copy-bit is always reset, since we assume it has been modified. The particular rules for arithmetic and logic operations are shown in Table 5.

Table 5: Rules for arithmetic operation and logic operation

Copy bit of a	Copy bit of b	Security class of the result	Copy bit of the result
cp	cp	$\underline{\mathrm{PC}}$	$\overline{cp}$
cp	$\overline{cp}$	$\underline{b}$	$\overline{cp}$
$\overline{cp}$	cp	$\underline{a}$	$\overline{cp}$
$\overline{cp}$	$\overline{cp}$	$\underline{a} \oplus \underline{b}$	$\overline{cp}$

3.3.6.3 Rules for comparison

In the C programming language, a comparison expression gets a Boolean expression as the result. Take $a > 3$ as an example, if the value of *a* is less than or equal to 3, the result of the comparison is 0, which indicates false in the C-Language. Otherwise, if *a* is greater than 3, the result is 1, which means true. For the comparison $a > c$, to make sure we maintain the proper tag, the result of the comparison should get the appropriate security class. Similar to the rules for arithmetic and logic operations, the security class of the result depends on the copy-bits and the security classes of *a* and *b*. The copy-bit of the result of comparison is always reset since it is a new value. The explicit rules for comparing x and y are shown in Table 6.

Approved for Public Release; Distribution Unlimited.

Table 6: Rules for comparison

Copy bit of x	Copy bit of y	Security class of the result	Copy bit of the result
cp	cp	$\underline{PC}$	$\overline{cp}$
cp	$\overline{cp}$	$\underline{y}$	$\overline{cp}$
$\overline{cp}$	cp	$\underline{x}$	$\overline{cp}$
$\overline{cp}$	$\overline{cp}$	$\underline{x} \oplus \underline{y}$	$\overline{cp}$

3.3.6.4 Information flow rules for assignment statement

For assignment statements, such as *y = x;*, we need to check whether the information is allowed to flow from *x* to *y*. In the lattice model of information flow, the assignment statement of *y = x*; requires the relation $\underline{x} \leq \underline{y}$ between *x* and *y* to ensure confidentiality. In addition, the security class of the current thread is also important to ensure integrity. We do not want unauthorized copying or modification of data. Therefore we require $\underline{y} \leq \underline{PC}$ for integrity. The value *x* can be a variable, a constant, a complex expression, etc. The rules for assignment statement (y = x;) are:

- When the copy-bit in the tag of *x* is cp, then the tag manager checks the relation of $\underline{y}$ and $\underline{PC}$. If $\underline{y} \leq \underline{PC}$, the assignment statement is allowed, otherwise, *x* is not allowed to be assigned to *y*. The tag of *x* is copied to *y*'s tag.
- When the copy-bit in the tag of *x* is $\overline{cp}$, then the tag manager checks the relation of $\underline{y}$ and $\underline{x} \oplus \underline{PC}$. If $\underline{x} \leq \underline{y} \wedge \underline{y} \leq \underline{PC}$, the information flow is allowed to flow from *x* to *y*, otherwise, *x* is not allowed to be assigned to *y*. The tag of *y* is unchanged.

If an assignment is not allowed, the hardware will generate a security exception and execute code in a specified exception handler.

When performing an assignment statement (*y = x;*), *x* might be the result of an arithmetic operation or a result of several arithmetic operations. We use our rules for arithmetic and logic operations (Section 3.3.6.2) to define the tag of the expression that *x* represents. Therefore, the information flow check can be done by checking the relation between the tag of *y* and the result of the operation (*x*).

Note that the check of $\underline{y} \leq \underline{PC}$ seems to violate the traditional security "no write down" model of Bell-Lapadula [Bell75], however, it satisfies the integrity controls of Biba [Biba77].

3.3.6.5 Information flow rules for if statement

For the if statement, such as *if(expression) commands;* and *if(expression) commands1; else commands2;*, we need to check whether the *expression* can be read by code, performing this if statement. To ensure that the *expression* can be read by code, that is, using the if statement, the relation between $\underline{PC}$ and $\underline{\text{expression}}$ needs to be checked before running the if statement. Therefore, the rule for if statement is:

Approved for Public Release; Distribution Unlimited.

- If expression ≤ PC, then the statement is allowed.

The *expression* can be a variable, a constant, a result of comparisons etc. The tag of the *expression* is calculated using the rules for comparison operations (Section 3.3.6.3) first, and then follows the information flow rules for the if statement to ensure proper information flows. The commands in the then and else statements can be controlled under other specific rules, such as rules for assignment statement.

3.3.6.6 Information flow rules for while statement

For the while statement, such as *while(expression) commands;*, we need to check whether the *expression* can be read by code performing the while statement. Similar to the rule for the *if* statement, to ensure the *expression* can be read by code using the while statement, the relation between PC and expression needs to be checked before executing the while statement. The rule for while statement is:

- If expression ≤ PC, then the statement is allowed.

The *expression* can be a variable, a constant, or a result of comparisons; its tag is also set using the appropriate rules. Whether or not the other commands in the while statement can be executed are controlled under other specific rules. Different from the if statement, the while statement body could be a loop of the commands if the *expressions* are true. Therefore, every time that while statement checks the Boolean value of the expression, the information flow rules for while statement should be used.

```
int start = 0x1;
int copy = 0;
int i=0;
int *value_to_steal = 0xABCDEF ; // user chosen addr

while (i<32) {
  if(*value_to_steal & start) copy = copy | start;
  start = start <<1;
  i=i+1;
}
```

Figure 17: Code to Copy a 32-bit Value

The reason we have the expression checking rules for if and while statements is demonstrated in Figure 17. This code steps through the bits of a user chosen address, *value_to_steal* and copies the bits to the variable copy. This is not a direct copy, but an implicit information flow that is prevented by our expression tag checks, the tag of **value_to_steal* is not less than the current PC tag.

Approved for Public Release; Distribution Unlimited.

3.3.6.7 Information flow rules for function calls

When performing a function call, we first check the function execution control rules (Section 3.3.4.3) If the function is allowed to be called by the caller, then we need to make sure the parameters passed to the function are allowed to be accessed by the code. Therefore, the security classes of the parameters and the security class of the caller, which is the security class of the current PC, are used in the check. The information flow rule for function calls is very similar to the rule for testing the expression in the while statement. The information flow rule for a function call like *foo(parameter1,parameter2);* is:

- For function calls, we first check the function execution control rules (Table 4). If the calling operation is allowed, and if $\underline{\text{parameter1}} \leq \underline{\text{PC}}$ and $\underline{\text{parameter}} \leq \underline{\text{PC}}$, then the function call is allowed to execute.

The tags of the parameter variables in the function will follow the expression and assignment rules. For example the call *foo(e1, e2)* will apply tagging rules for *parameter1 = e1* and *parameter2 = e2*.

Directive code, which needs to set the copy-bit of the tag of the return value, will call the directive *rtems_tag_copy* of the tag manager (Section 3.3.7) before returning the value. The *rtems_tag_copy* copies a value and sets the copy-bit of its tag, for return to the caller. This internal directive will bypass the standard tagging rules for assignment.

3.3.6.8 Other C constructs

We have not defined tagging rules for other C-Language constructs, such as the *for* loop or *switch* statement. These can be mapped to the constructs we have defined and will follow the rules of those constructs. We have also not defined rules for *break* and *continue*, since their only function is to change location of execution within the same code space. They do not have any information flow concerns for our tagging model.

3.3.7 Tag manager

We have developed a tagging scheme for an operating system that utilizes features of a security tagged architecture microprocessor being developed as part of a larger research project. The tag checks and tag propagation rules defined in the previous section, and further refined in Section 3.3, will be implemented in hardware. The hardware support is being implemented by Ed Suh's group at Cornell University. They are developing a coprocessor tag engine that will operate in parallel with a Leon3 processor[7]. For this project, we assume the existence of the tagging coprocessor and do not define its implementation.

To support operating system level tagging for RTEMS, we need to add a tag manager to the system. The purpose of the tag manager is to allow system initialization software to configure the initial tags of the system, to allow trusted software to set and modify tags, to allow the directives

[7] The Leon3 is based on the OpenSparc processor.

to set and validate the copy-bit, and to manage the tagging exceptions thrown by the hardware when a tag violation occurs. This section outlines the directives of the tag manager for the RTEMS system. Implementation of this manager is left for future work, waiting for completion of the supporting hardware or simulator. The directives of the tag manager are summarized below, the RTEMS APIs for these directives are provided in Appendix A.

Tag a section of memory: The following directives tag one or more words of memory as specified by the caller. These directives are internal functions only callable by directives or other internal functions. These calling directives can assign one of three tags to the specified memory block for itself or for the user calling the directive. For example if the caller is a manager4 acting on behalf of a user1, then the tag can be (manager4, manager4) or (user1, manager4) or (user1, user1).

- rtems_tag_partition. This directive tags every word in a specified RTEMS memory partition with a specified tag.
- rtems_tag_segment. This directive tags every word in a specified RTEMS memory segment of a specified memory region with a specified tag.
- rtems_tag_word. This directive tags a specified memory word with a specified tag.

Copy-bit control: The following tag manager directives support use of the copy bit.

- rtems_tag_copy. This directive copies the value and tag from the specified source expression to the specified destination address, and sets the copy-bit in the destination.
- rtems_tag_validate_copybit. This directive validates that the copy-bit is set and that the tag of the input is the same as the tag of the calling thread. If validation fails, a tagging exception is thrown.
- rtems_tag_release. This directive allows downgrading the tag of a specified value *x*, so that it may be copied to the specified target, *y*, where the assignment rule would normally have prohibited the assignment. The tag of *x* will be set to the tag of the calling program (effectively treating *x* as a constant) for the copy.

Tag engine control: The following directives are used to directly control the operation of the hardware tag engine. Each of these is a highly privileged directive and will be callable only by SCORE.

- rtems_tag_enable. This directive enables the tagging mechanism (this may end up being a restricted internal function that is part of the initialization routines, but currently in place for experimentation).
- rtems_tag_disable. This directive disables the tagging mechanism (a dangerous function, and may be restricted to just shutdown phases of RTEMS, but currently in place for experimentation).

Approved for Public Release; Distribution Unlimited.

- rtems_tag_set_handler. This directive sets the exception handling function that is executed when the hardware tag engine detects a tagging violation.
- rtems_tag_set_lattice. This directive sets rules to form a Lattice based on both of the Owner field and the Code-space field in the tag. (We may remove this if we hard code the tag format as specified in Table 3).
- rtems_tag_initialize. This directive initializes the tag manager, and tags all of the code space with appropriate tags. For this to work correctly, the system will have to have configuration information that indicates the tags associated with each function.

3.3.8 Add security tags to RTEMS code

Shown in Figure 18 we use a small segment of user code, *rtems_task_create* and directive code *rtems_task_delete* to illustrate how our security tagging scheme works.

- Starting from user code (lines 1-18), because user code is running, the tag for the PC is (user1, user1, $\overline{cp}$).
- On lines 2-5, user code creates four variables *Task_id*, *Task_id2*, *Task_name* and *Task_name2*. These variables should be tagged (user1, user1, $\overline{cp}$) to indicate that they are created by the user1 in user1.
- In the *Init* function, *status* is given the tag (user1, user1, $\overline{cp}$) on line 7.
- Then, on lines 9-11 (the function call), we check permissions for the user to make the call to *rtems_task_create*, using our function execution rules (Section 3.3.4.3). Since *rtems_task_create* is a directive of task manager, which should have tag (task manager, task manager, $\overline{cp}$) associated with its code, and the tag of the PC is (user1, user1, $\overline{cp}$), the function call is allowed. Then, the information flow rule for function calls (Section 3.3.4.3) will be used to check the accessibility of the parameters. All parameters are either local variables whose tags are PC's tag, or constants, therefore they are valid parameters. Since the function call is allowed, the tag of PC will be changed to (user1, task manager, $\overline{cp}$), when the system executes the directive code.
- In the directive code *rtems_task_create* (lines 22-38), the values of name, *stack_size*, *initial_priority*, *inital_modes attribute_set* and **id* are all from the user, consequently they still have the tags (user1, user1, $\overline{cp}$) associated with them. The pointer, *the_thread*, is declared in task manager for the user1, so it has tag (user1, task manager, $\overline{cp}$).
- On line 32, the internal routine *_RTEMS_tasks_Allocate* is called. It is a task manager internal function, so according to our function execution control rules, *the_thread* which has tag (user1, task manager, $\overline{cp}$), it is allowed to call its internal function.
- On line 34, the directive of tag manager, *rtems_tag_copy*, is used to copy the value and tag of *the_thread->Object_id* to *id* and also sets the copy-bit of the tag. Therefore, **id* is tagged (user1, task manager, *cp*). The settings of the copy-bit allow the user to receive an authenticated ID and reuse it later.

- When returning from directive to user code, the tag of the PC will change back to (user1, user1, $\overline{cp}$) and *Task_id* will get the tag (user1, task manager, cp).
- Similar to the code on lines 9-11, after executing the code of lines 12-14, *Task_id2* gets the tag (user1, task manager, cp).
- On line 16, the user calls the *rtems_task_delete* directive to delete the task associated with *Task_id*. Because *Task_id* has tag (user1, task manager, cp), the copy-bit is set. Therefore the *rtems_task_delete* directive trusts the value of the *Task_id* after checking it on line 44, and deletes the task for the user1.
- For *Task_id2 + 1*, *Task_id2* has tag (user1, task manager, cp), and 1 has the PC's tag ((user1, user1, $\overline{cp}$)). According to the rules for arithmetic operation, the copy-bit of the *Task_id2*'s tag is set, so the tag of the result value will be 1's tag ((user1, user1, $\overline{cp}$)) and the copy-bit will be reset to $\overline{cp}$. It means that it is a data from user1, but not task manager any more. On line 17, the user passes the modified *Task_id2* to the *rtems_task_delete* directive. The *rtems_task_delete* directive on line 44 calls *rtems_tag_validate_copybit* and will not trust the parameter because the copy-bit in the tag is not set. Consequently the directive will not perform the deletion of the task associated with *Task_id2* for the user1.
- When *rtems_task_delete* calls *_Thread_Get*, on line 47, the *_Thread_Get* is a SCORE function and it has tag (thread, thread, $\overline{cp}$). The *_Thread_Get* function can be called by the directive (Manager class) using our function execution control rules and the parameter is allowed to be accessed under our information flow rule for function calls.
- Then whether or not the return value of *_Thread_Get* function is allowed to be assigned to *the_thread* is checked. Since *the_thread* is local to *rtems_task_create* and *_Thread_Get* returns a value for the calling function, its tag will be (user1, task manager, $\overline{cp}$). Then the calls to *_Thread_Close*, which is a SCORE function and *_RTEMS_tasks_Free*, which is a task internal function, are allowed.

```
/* User code: */
   rtems_id Task_id;
   rtems_id Task_id2;
   rtems_name Task_name;
   rtems_name Task_name2;
   rtems_task Init(rtems_task_argument argument)
   {rtems_status_code status;
    ...
    status = rtems_task_create(
    Task_name, 1, RTEMS_MINIMUM_STACK_SIZE * 2,
       RTEMS_DEFAULT_MODES,
    RTEMS_DEFAULT_ATTRIBUTES, &Task_id);
    status = rtems_task_create(
    Task_name2, 1, RTEMS_MINIMUM_STACK_SIZE * 2,
       RTEMS_DEFAULT_MODES,
    RTEMS_DEFAULT_ATTRIBUTES, &Task_id2);
    ...
    status = rtems_task_delete(Task_id);
    status = rtems_task_delete(Task_id2 + 1);
   }


/* Directive code: */
rtems_status_code rtems_task_create(
    rtems_name              name,
    rtems_task_priority     initial_priority,
    size_t                  stack_size,
    rtems_mode              initial_modes,
    rtems_attribute         attribute_set,
    Objects_Id             *id
    )
    {register Thread_Control *the_thread;
     ...
     the_thread = _RTEMS_tasks_Allocate();
     ...
     rtems_tag_copy(the_thread->Object_id, id);
     ...
     return RTEMS_SUCCESSFUL;
    }

rtems_status_code rtems_task_delete(Objects_Id id)
    {register Thread_Control *the_thread;
     Objects_Locations         location;
     Objects_Information      *the_information;
     ...
     if(!rtems_tag_validate_copybit(id)){
     return RTEMS_ACCESS_VIOLATION;
     }else{
     the_thread = _Thread_Get(id, &location);
     ...
     _Thread_Close(the_information, the_thread);
     _RTEMS_tasks_Free(the_thread);
     ...
     return RTEMS_SUCCESSFUL;
     ...
    }}
```

Figure 18: Sample C code 3

3.4 IMPLEMENTATION OF TAGGING SCHEME AT THE ASSEMBLY-LANGUAGE LEVEL

Section 3.3 introduced our tagging scheme and its implementation at the C programming language level. However, the higher-level abstraction of the C-Language ignores important implementation details that will affect the security of our tagging scheme. As a result Section 3.4 provides a first pass analysis of an assembly-language level implementation of the tagging scheme. We provide rules for assembly language instructions that have direct C-Language counter parts and discuss concerns for future work, such as trap (i.e., interrupt) handling.

This project will implement tagging on a Scalable Processor ARChitecture (SPARC) processor. Section 3.4 highlights important features of the SPARC processor, and assembly-level tagging rules. For example, in SPARC, at any time, the user can access 8 *in* registers, 8 *local* registers, 8 *out* registers and 8 *global* registers. SPARC uses a register window to help pass parameters to subroutines and return values to the caller. SPARC stores the PC, parameters, return value, frame pointer, and stack pointer in specific registers within the window. For that reason our tagging rules must take this style of parameter passing into account. During our study of SPARC instructions, we divided them into four groups and found that the information flow control rules we designed in the C programming language needed to be refined to fit into the SPARC architecture. Some of the rules for SPARC instructions are similar to the rules for C-Language, but some of the security features require the implementation of new instructions. What is more, at the assembly language level there are more things that we need to consider, such as the runtime stack, use of registers, and the program counter (PC). We used *sparc-rtems-gcc* to generate several simple assembly code examples to help illustrate how this all comes together (Section 3.5.2).

The remainder of Section 3.4 is organized as follows: in Section 3.4.1, the concepts of the SPARC architecture, register window and memory stack are briefly introduced. In Section 3.4.2, we discuss the SPARC instructions and divide the instructions into four groups and design tagging rules for them. In Section 3.4.3, some concerns about the security of stack are discussed. Section 3.4.4 introduces the proposed representation of memory tags. Lastly, we generate some assembly code and analyze the code in Section 3.4.5.

3.4.1 SPARC architecture

SPARC has an instruction set architecture that is derived from a reduced instruction set computer (RISC) lineage [SPARC]. The difference between SPARC and RISC is that SPARC provides flexible register window management by using separate instructions for window management, function call and return. The main features of SPARC are:

- The address space is 32-bit, linear.
- All instructions are 32 bits, and the number of instructions is reduced.
- A separate floating-point register file.
- A large register window. A program can see a 24 register window and 8 global registers at any time.
- The processor fetches the next instruction after a delayed control-transfer instruction.

- The architecture defines a coprocessor instruction set.
- It has instructions for a synchronizing multiprocessor.
- Trap causes an allocation of a new register window.

3.4.1.1 SPARC register set and register windows

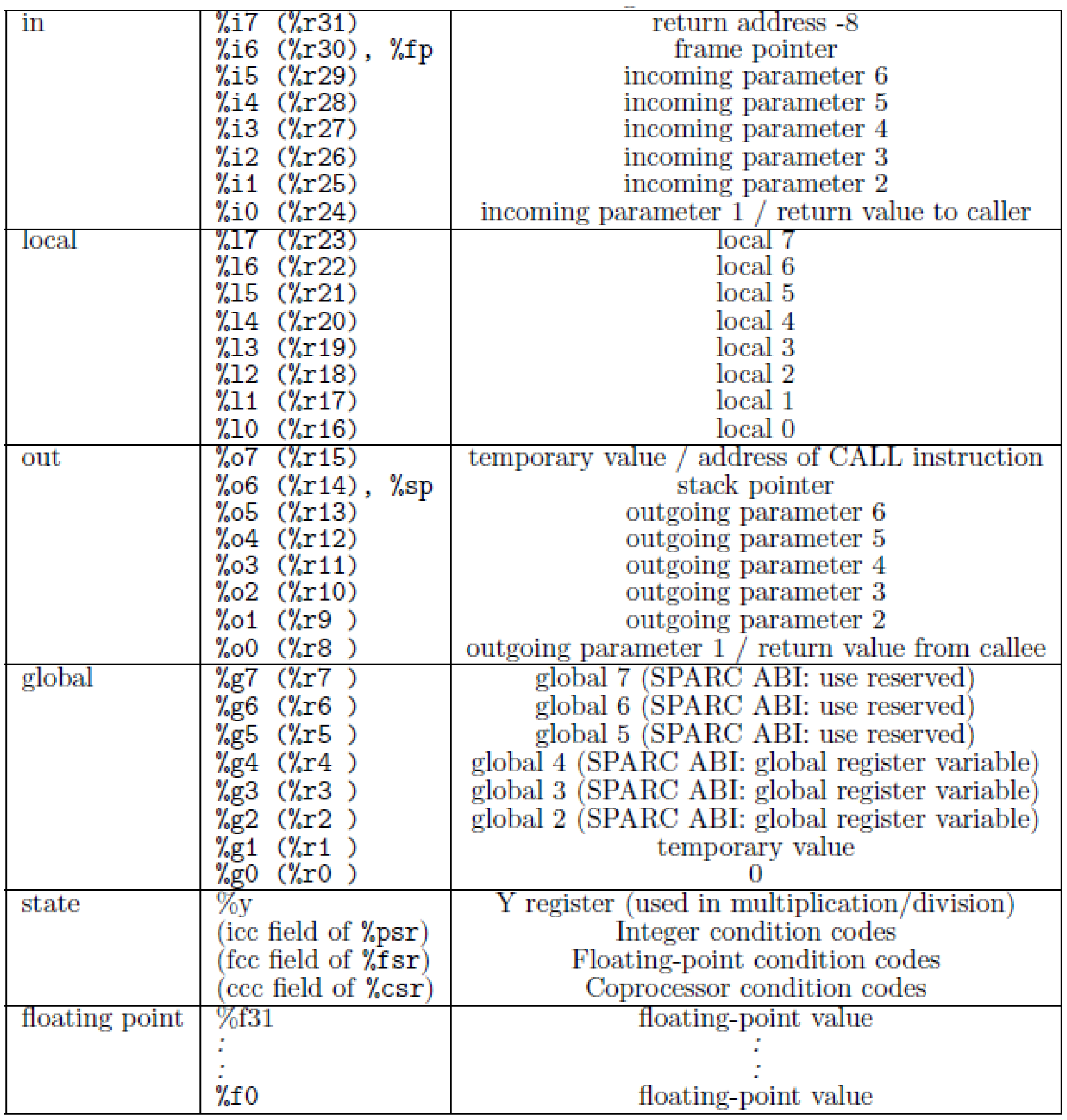
At a given time, an instruction can see 8 *global* registers and a register window which contains 24 registers: 8 *in* registers, 8 *local* registers and 8 *out* registers. Figure 19 (adapted from SPARC International Inc. [SPARC]) shows the register set that can be seen by user. In and out registers are used for passing parameters and getting return vales, because the caller's *out* registers become the callee's *in* register when there is a normal function call[8]. Registers *%i0-%i7* are in registers. The first six in registers are used to store incoming parameters. In addition, *%i0* is also used to store the return value to the caller, *%i6* is for storing the frame pointer (*%fp*) and *%i7* is for storing the return address. Registers *%l0-%l7* are local registers that are mostly used for temporary values. The current PC and the next PC (nPC) are copied to *%l1* and *%l2* when a trap occurs. Registers *%o0-%o7* are the out registers. Among these registers, *%o0-%o5* are used to store the parameters being passed to a called subroutine, *%o6* stores the stack pointer, and *%o7* stores a temporary value or the return address. The *CALL* instruction writes its own address into *%o7* and the address is then used as the basis for the return address for the callee. The global registers are *%g1-%g7* and have global scope. The *%g0* register has the hardwired value of zero; any write to it has no effect.

In the SPARC architecture, partially overlapping windowed integer registers are provided. A window contains 8 *in* registers, 8 *local* registers, and 8 *out* registers. The current window pointer (CWP) indicates the current window and walks a circular buffer of windows[9]. The execution of instruction *SAVE* or trap decreases the CWP by 1 and instruction *RESTORE* or *RETT* increases the window by 1. The overlap of windows allows each window to share its *in* and *out* registers with adjacent windows. As shown in Figure 20 (adapted from SPARC International Inc. [SPARC]), when the CWP decreases by one, the out registers of the previous register window become the *in* registers of the new register window and the caller's stack pointer (*%sp*) becomes the current procedure's frame pointer (*%fp*). When executing *RESTORE* or *RETT*, the CWP increases by one and the window moves back to the previous window. The *local* registers are unique to each window. A return value is stored in *%i0*, therefore, after moving the register window with a *RESTORE*, the caller can access the return value that is stored in its *out* register *%o0*, which was callee's *in* register *%i0*. Window overflows and underflows cause a trap to the operating system.

[8] The CALL instruction does not cause this register mapping, rather the SAVE instruction slides the register window -- allowing more flexibility for the compiler, but decoupling parameter passing from the actual function call. The *RESTORE* instruction is used to slide the window back before a return.

[9] The Leon3 processor supports 32 register windows.

Approved for Public Release; Distribution Unlimited.



in	%i7 (%r31)	return address -8
	%i6 (%r30), %fp	frame pointer
	%i5 (%r29)	incoming parameter 6
	%i4 (%r28)	incoming parameter 5
	%i3 (%r27)	incoming parameter 4
	%i2 (%r26)	incoming parameter 3
	%i1 (%r25)	incoming parameter 2
	%i0 (%r24)	incoming parameter 1 / return value to caller
local	%l7 (%r23)	local 7
	%l6 (%r22)	local 6
	%l5 (%r21)	local 5
	%l4 (%r20)	local 4
	%l3 (%r19)	local 3
	%l2 (%r18)	local 2
	%l1 (%r17)	local 1
	%l0 (%r16)	local 0
out	%o7 (%r15)	temporary value / address of CALL instruction
	%o6 (%r14), %sp	stack pointer
	%o5 (%r13)	outgoing parameter 6
	%o4 (%r12)	outgoing parameter 5
	%o3 (%r11)	outgoing parameter 4
	%o2 (%r10)	outgoing parameter 3
	%o1 (%r9)	outgoing parameter 2
	%o0 (%r8)	outgoing parameter 1 / return value from callee
global	%g7 (%r7)	global 7 (SPARC ABI: use reserved)
	%g6 (%r6)	global 6 (SPARC ABI: use reserved)
	%g5 (%r5)	global 5 (SPARC ABI: use reserved)
	%g4 (%r4)	global 4 (SPARC ABI: global register variable)
	%g3 (%r3)	global 3 (SPARC ABI: global register variable)
	%g2 (%r2)	global 2 (SPARC ABI: global register variable)
	%g1 (%r1)	temporary value
	%g0 (%r0)	0
state	%y	Y register (used in multiplication/division)
	(icc field of %psr)	Integer condition codes
	(fcc field of %fsr)	Floating-point condition codes
	(ccc field of %csr)	Coprocessor condition codes
floating point	%f31	floating-point value
	:	:
	:	:
	%f0	floating-point value

Figure 19: SPARC registers

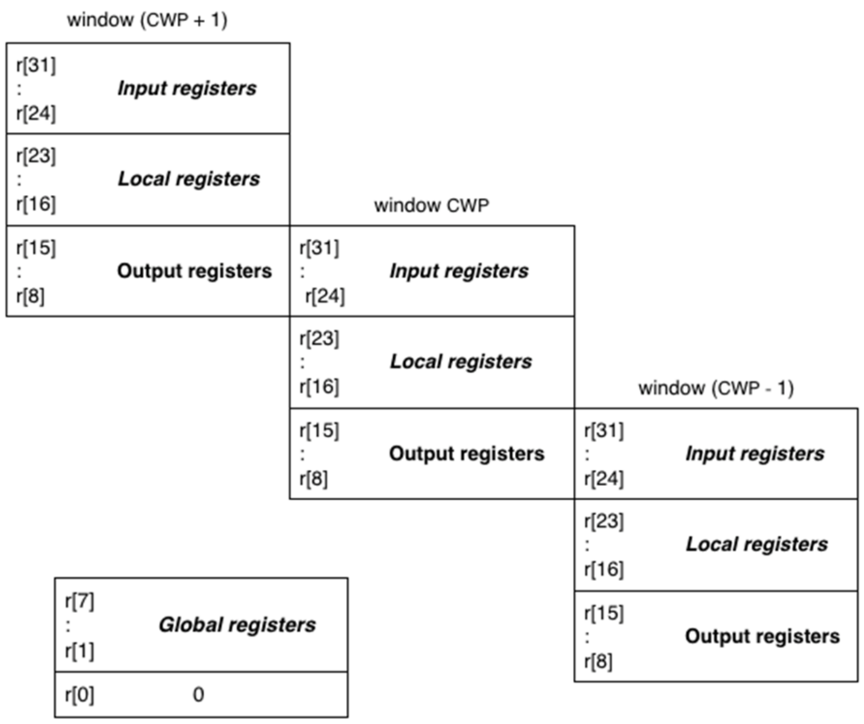


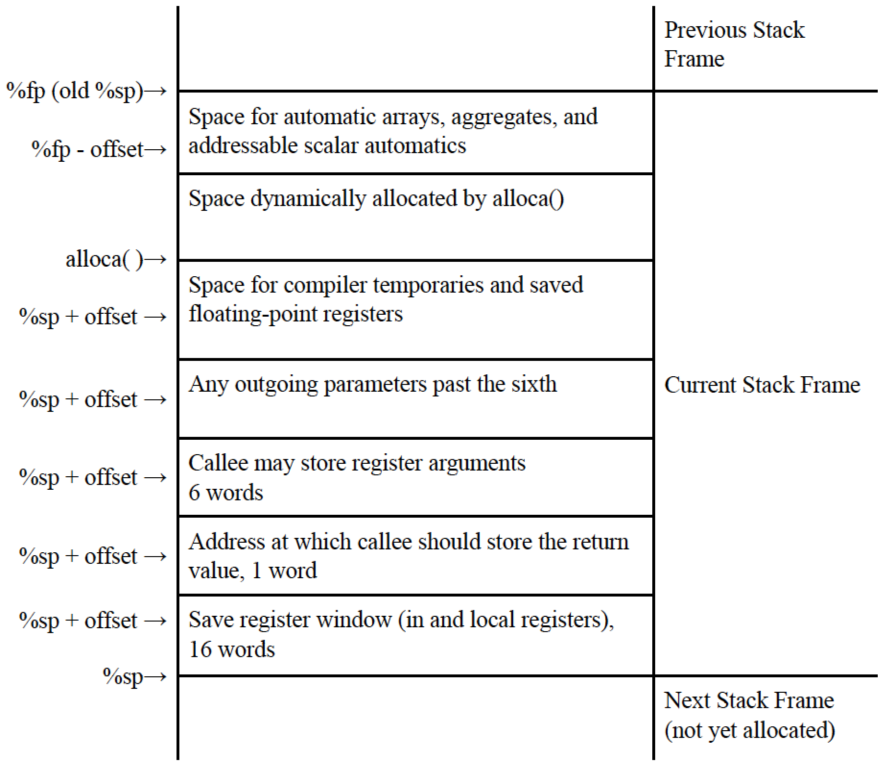
Figure 20: Change of the register window

3.4.1.2 SPARC stack

Normally in SPARC, a stack frame, which is contiguous space on the memory stack, is allocated for each procedure. The memory stack grows down toward decreasing memory addresses. As shown in Figure 20 (adapted from SPARC International Inc.[SPARC]), starting at the *%sp* and going up, there are 16 words for saving the *in* and *local* registers for the current procedure. Then, there is one word for the extra return value and six words for saving callee's parameters. After that, space is allocated for other things, such as temporary vales generated by compiler, outgoing parameters and memory allocated by *alloca()* function.

3.4.2 Tagging rules for SPARC instructions

In Section 3.3.6 we introduced our tagging scheme from the point of view of the C programming language. This section explains the implementation of our tagging scheme at the assembly language level. In this section, we classify the SPARC instructions into smaller groups and refine the tagging rules for C-Language for each group of instructions.



%fp (old %sp)→
%fp - offset→
alloca( )→
%sp + offset →
%sp + offset →
%sp + offset →
%sp + offset →
%sp + offset →
%sp→
Previous Stack Frame
Space for automatic arrays, aggregates, and addressable scalar automatics
Space dynamically allocated by alloca()
Space for compiler temporaries and saved floating-point registers
Any outgoing parameters past the sixth
Callee may store register arguments 6 words
Address at which callee should store the return value, 1 word
Save register window (in and local registers), 16 words
Current Stack Frame
Next Stack Frame (not yet allocated)

Figure 21: Memory stack of SPARC

The SPARC instructions can be classified into four groups:

- Group 1 defines the 4 SETHI and branch instructions.
- Group 2 defines the 3 CALL related instruction.
- Group 3 defines the 51 arithmetic, logical, shifting and instructions not in other groups with the 20 floating point instructions.
- Group 4 defines the 51 memory instructions.

The tagging rules for SPARC instructions are discussed in detail in the following sections.

3.4.2.1 Rules for branch instructions

The branch instructions are shown in Table 7. Branch instructions are either unconditional branches or conditional branches. Unconditional branch instructions are *BA* (branch always), which means always execute the branch, and *BN* (branch never), which means never execute the branch (effectively a *NOP*). Conditional branches are based on the condition codes, which are updated by compare instructions, to decide whether or not make the branch. Therefore, whether or not the current running thread can access the condition code should be checked. For the execution of the target code, a check is needed to make sure the current thread is allowed to execute (jump to) the target code. Consequently, for branch instructions such as *Bicc address*}, the address and PC must be checked to ensure the jump is appropriate. Since this is a jump, we want to make sure we don't change code spaces. For the branch instructions, we must stay in the same code space, therefore, the Code-space(address)[10] and Code-space(PC) must be checked to make sure the branch is allowed to execute the target code. In addition, three bits are used in the Control-bit field of a tag to show the memory type of the tagged code or data. It follows that we can check and ensure that the target address is a executable code memory, but not data memory or stack memory (Section 3.3.3.2 for the Control-bit field of tags and the memory types). The rule for branch instructions is:

The if [condition code] ≤ PC, and Code-space(address) = Code-space(PC), and the target address is an executable code memory then the branch is allowed. Otherwise, throw an exception.

As with the C-code, if any instruction is not permitted, the hardware generates a security exception.

Table 7: Four Implemented SETHI and Branch instructions

Opcode	Name
SETHI	Set High 22 bits of r Register
Bicc	Branch on integer condition codes
FBfcc	Branch on floating-point condition codes
CBccc	Branch on coprocessor condition codes

3.4.2.2 Rules for call related instructions

Table 8: Four Implemented and One new Call-related functions

Opcode	Name
CALL	Call and Link
JMPL	Jump and Link
RETT	Return from Trap
Ticc	Trap on integer condition codes
RET	New instruction to return from function call

The four call related instructions are *CALL*, *JMPL*, *RETT* and *Ticc*. When making a function call (*CALL address*), the tag system must check whether or not the function is allowed to be called by the current program using our function execution control rules. Our function execution control rules specify if the current program is not allowed to call the function, then the call will not be executed. If it is allowed under our rules and the target code is the entry point of an executable function, then the call is allowed. The call will save the value of PC into *%o7*, PC into %o7, set the copy-bit of *%o7*, update the PC and tag of the PC, and then start execution of

[10] Code-space(address) refers to just the Code-space field of the addresses tag.

the function. The tag of the updated PC will follow the function execution control rules. The rule for *CALL* instruction is:

- If the function call is allowed under the function execution control rule, and if the target code is tagged as entry point to an executable function, then the *CALL* instruction is allowed.

In SPARC, the *CALL* instruction is a shorthand notation for *JMPL address, %o7*. For our purposes we require a unique call instruction.

The *JMPL* instruction is used to save the current PC in a specified register and then jump to any specified address. There are two usages of *JMPL*. The first is *JMPL* address, *%g0* which copies the current PC to *%g0* before making the jump. However, because *%g0* always has value 0 in it and cannot be overwritten, the write to it has no effect. As a result this is a true jump with no expected return. As with branch instruction, we need to make sure that the current running program has the permission to jump and execute the target code, for that reason the security class of PC and the target address must be compared. The rule for the first usage of the *JMPL* instruction is:

- if Code-space(address) = Code-space(PC), and the target address is an executable code memory then the JMPL is allowed.

The second usage, JMPL address, *%o7* will write the current PC to *%o7*. Since SPARC uses *JMPL %o7+8, %g0* to jump back according to the return address stored in *%o7*, a check is needed to ensure the return address has not been modified. However, *JMPL* has many other usages, so we cannot give *JMPL* such a restrictive rule. Consequently we require the implementation of a new instruction, *RET*, which is used for return from subroutines. The format of *RET* instruction could be *RET %o7+8, %g0* (SPARC specifies a *RET* instruction that is just a shorthand notation for *JMPL %o7+8, %g0*, we will replace this with a separate instruction). For return from a subroutine, if the copy-bit of the tag of *%o7* is not set, then the return is not allowed. If the copy-bit is set, which ensures that the return address was not modified, then we need to check the PC and the return address stored in *%o7* and also the $\underline{\text{PC}}$ and $\underline{\text{\%o7}}$. The check is to ensure that the return address has not been modified. The rule for *RET* is:

- When the copy-bit of *%o7* is not set, the *RET* instruction is not allowed. If the copy-bit of *%o7* is set, and Code-space([%o7+8]) = Code-space(%o7) and the target address ([%o7+8]) is an executable code, then the *RET* is allowed, with the new PC = %o7+8, and the security class of the tag of PC is $\underline{\text{\%o7}}$.

RETT is used to return from a trap handler. *Ticc* is used to generate a trap to the trap handler based on an integer condition code. For this initial tagging system, we apply the same rules we apply to *CALL* and *RET* for these instructions, with the understanding that they will have to be modeled and possibly changed for a fully secure system.

3.4.2.3 Rules for arithmetic, logic and shifting instructions

Table 9: Twelve Implemented Logical Operations

Opcode	Name
AND (ANDcc)	And (and modify icc)
ANDN (ANDNcc)	And Not (and modify icc)
OR (ORcc)	Inclusive-Or (and modify icc)
ORN (ORNcc)	Inclusive-Or Not (and modify icc)
XOR (XORcc)	Exclusive-Or (and modify icc)
XNOR (XNORcc)	Exclusive-Nor (and modify icc)

Table 10: Three Implemented Shifting Instructions

Opcode	Name
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetic

Table 11: Twenty One Implemented Arithmetic Instructions

Opcode	Name
ADD (ADDcc)	Add (and modify icc)
ADDX (ADDXcc)	Add with Carry (and modify icc)
TADDcc (TADDccTV)	Tagged Add and modify icc (and Trap on overflow)
SUB (SUBcc)	Subtract (and modify icc)
SUBX (SUBXcc)	Subtract with Carry (and modify icc)
TSUBcc (TSUBccTV)	Tagged Subtract and modify icc (and Trap on overflow)
MULScc	Multiply Step (and modify icc)
UMUL (UMULcc)	Unsigned Integer Multiply (and modify icc)
SMUL (SMULcc)	Signed Integer Multiply (and modify icc)
UDIV (UDIVcc)	Unsigned Integer Divide (and modify icc)
SDIV (SDIVcc)	Signed Integer Divide (and modify icc)

For arithmetic instruction (shown in Table 11) logic instructions (shown in Table 9) and shifting instructions (shown in Table 10) the rules are similar to the rules for arithmetic and logic statements (Section 3.3.6.2) that we use in the C-Language. For example, the instruction *ADD %g2*, *%g1*, *%g3* adds the value in *%g2* to the value in *%g1*, then stores the result in *%g3*. According to the rules in C-Language, the copy-bit in the tags of the value in *%g1* and *%g2* need to be checked, and the result value stored in *%g3* has a new tag associated with it. Whether or not the result can be stored in *%g3* is not checked, instead we use registers as temporary storage. Therefore, the rules for arithmetic, logic, and shifting instructions can follow the rules for arithmetic and logic expression in C-Language. Taking *ADD %g2, %g1*, *%g3* as an example, the particular rules for arithmetic, logic, and shifting instructions are shown in Table 12.

Table 12: Rules for arithmetic, logic and shifting instructions

Copy-bit of %g2	Copy-bit of %g1	Security class of %g3	Copy-bit of %g3
cp	cp	PC	$\overline{cp}$
cp	$\overline{cp}$	%g1	$\overline{cp}$
$\overline{cp}$	cp	%g2	$\overline{cp}$
$\overline{cp}$	$\overline{cp}$	%g1 ⊕ %g2	$\overline{cp}$

For those instructions that modify the condition code, such as *ANDcc*, *ORcc*, *SUBcc*, and *UMULcc*, the tag of the condition code will be identical to the tag of the result. For example, after executing the instruction *ADDcc %g2, %g1, %g3*, the condition code's tag will be the tag of *%g3*.

If %g1 is an immediate value, then %g1's tag is PC's tag with the copy-bit not set.

3.4.2.4 Rules for load and store instructions

ST instructions are shown in Table 13. They are used to store the value from a register to a memory space. The rule for checking whether or not the value of the register is allowed to be stored in that memory location can be ensured by our assignment rules in the C-Language (Section 3.3.6.4). For example, the instruction *ST %g1, [%fp-12]* stores the content of *%g1* to memory space *[%fp-12]*. This can be mapped to the assignment statement *y = x;*, which means to store the value of *x* to the memory space of *y*. Therefore, the assignment rules in C-Language can be used for store instructions. The rules for store instructions (*ST %g1, [%fp-12]*) are:

- When the copy-bit in the tag of *[%fp-12]* is $\overline{cp}$ and the copy-bit in the tag of *%g1* is $\overline{cp}$, if $\underline{[\%fp\text{-}12] \leq PC}$ and $\underline{\%fp} \leq \underline{PC}$ and *[%fp-12]* is writable data memory, the store instruction is allowed. The tag of the *[%fp-12]* is unchanged.
- When the copy-bit in the tag of *[%fp-12]* is $\overline{cp}$ and the copy-bit in the tag of *%g1* is *cp*, if $\underline{[\%fp\text{-}12]} \leq \underline{PC}$ and $\underline{\%fp} \leq \underline{PC}$ and Owner(*%g1*) = Owner*([%fp-12]*) and *[%fp-12]* is writable data memory, the store instruction is allowed. The tag of the [%fp-12] is copied from the tag of *%g1*.
- When the copy-bit in the tag of *[%fp-12]* is *cp* and the copy-bit in the tag of *%g1* is $\overline{cp}$, if Owner(*[%fp-12]*) ≤ Owner(PC) and $\underline{\%fp} \leq \underline{PC}$ and *[%fp-12]* is writable data memory, the store instruction is allowed. The Code-space of the tag of the *[%fp-12]* will be reset to the Owner field of the tag and the copy-bit will be unset.
- When the copy-bit in the tag of *[%fp-12]* is *cp* and the copy-bit in the tag of *%g1* is *cp*, if Owner(*[%fp-12]*) ≤ Owner(PC) and $\underline{\%fp} \leq \underline{PC}$ and Owner(*%g1*) = Owner(*[%fp-12]*) and *[%fp-12]* is writable data memory, the store instruction is allowed. The tag of the *[%fp-12]* is changed to the tag of *%g1*.
- If *[%fp-12]* is writable stack memory, then write is allowed and tag of *[%fp-12]* is tag of *%g1*.
-

Table 13: Implemented Store Instructions

Opcode	**Name**
STB (STBA)	Store Byte (into Alternate space)
STH (STHA)	Store Halfword (into Alternate space)
ST (STA)	Store Word (into Alternate space)
STD (STDA)	Store Doubleword (into Alternate space)
STF	Store Floating-point
STDF	Store Double Floating-point
STFSR	Store Floating-point State Register
STDFQ	Store Double Floating-point deferred-trap Queue
SWAP (SWAPA)	Swap r Register with Memory (in Alternate space)
Not implemented	
STC	Store Coprocessor
STDC	Store Double Coprocessor
STCSR	Store Coprocessor State Register
STDCQ	Store Double Coprocessor deferred-trap Queue

Table 14 shows the load instructions in SPARC architecture. Load instructions are used to load a value from a memory space and store it to a register. For example, the instruction *LD [%fp-16], %o1* loads the content of the memory space *[%fp-16]* to *%o1* register. A check of whether the current program can read and use the value is needed. What is more, we need to check that the current running thread can access the value of *%fp*. Therefore the security classes of PC and the tag of the value and the tag of the *%fp* need to be checked. The load instruction is allowed when $\underline{\text{[\%fp-16]}} \le \underline{\text{PC}}$, where $\underline{\text{[\%fp-16]}}$ denotes the security class of the tag of the value stored in *[%fp-16]*. The rules for load instructions enforce these checks of expressions and parameters. The rules for load instructions (*LD [%fp-16], %o1*) are:

- If the copy-bit in the tag of *[%fp-16]* is *cp*, and $\underline{\text{\%fp}} \le \underline{\text{PC}}$, and Owner(*[%fp-16]*) ≤ Owner(PC), and *[%fp-16]* is readable memory, then the load is allowed and the tag of the data will be copied to the register's tag.
- If the copy-bit in the tag of *[%fp-16]* is $\overline{cp}$, and if $\underline{\text{[\%fp-16]}} \le \underline{\text{PC}}$ and $\underline{\text{\%fp}} \le \underline{\text{PC}}$, and *[%fp-16]* is readable memory, then the load instruction is allowed and the tag of the data will be copied to the register's tag.

Approved for Public Release; Distribution Unlimited.

Table 14: Seventeen Implemented Load instructions

Opcode	Name
LDSB (LDSBA)	Load Signed Byte (from Alternate space)
LDSH (LDSHA)	Load Signed Halfword (from Alternate space)
LDUB (LDUBA)	Load Unsigned Byte (from Alternate space)
LDUH (LDUHA)	Load Unsigned Halfword (from Alternate space)
LD (LDA)	Load Word (from Alternate space)
LDD (LDDA)	Load Doubleword (from Alternate space)
LDF	Load Floating-point
LDDF	Load Double Floating-point
LDFSR	Load Floating-point State Register
LDSTUB (LDSTUBA)	Atomic Load-Store Unsigned Byte (in Alternate space)
Not implemented	
LDC	Load Coprocessor
LDDC	Load Double Coprocessor
LDCSR	Load Coprocessor State Register

3.4.2.5 Rules for other instructions

The *SAVE* and *RESTORE* instructions are used to slide the register window between the caller and callee windows. It is possible but rare, to use these instructions without a corresponding subroutine call. The execution of the *SAVE* instruction causes the system to subtract one from the CWP. Therefore, the instruction automatically allocates a new window of registers and a new stack frame in the main memory. The out registers of the caller become the in registers of the callee. The *SAVE* instruction acts like an *ADD* instruction, which adds a number to *%sp*. The number indicates the size of the stack for the new function. The caller's stack pointer *%sp* automatically becomes the frame pointer *%fp* of the callee. The instruction *RESTORE* is used to restore the caller's window. The instruction adds one to CWP, therefore the callee's *in* registers become the caller's *out* registers.

Since the user can use *SAVE* and *RESTORE* instructions, we are concerned about the security of these windows. The user may use the *RESTORE* instruction to change the register window to get data used in the caller's subroutine. Therefore, we add one tag to every register window to indicate who created the register window. On every *SAVE* instruction, we copy the tag of the PC to the tag of the new window. By doing this, every created window has a tag associated with it.

On every *RESTORE* instruction, we check the tag of the caller's window. If the tag indicates that creator of the window is not the one who wants to restore (pop) the window, then the restore instruction is not allowed to execute. During traps and exceptions, we want to give the SCORE code the ability to manage the register window. As a result, the SCORE code has the permission to restore any window or register without any check of the window's tag. If the tag of the register window is identical to the tag of the PC, then the restore is allowed. Otherwise, the running thread is not the creator of the register window and cannot restore the register window.

Approved for Public Release; Distribution Unlimited.

Table 15: Other Instructions - Including 15 Special Purpose and 20 Floating Point

Opcode	**Name**
SAVE	Save callers window
RESTORE	Restore callers window
RDASR	Read Ancillary State Register
RDY	Read Y Register
RDPSR	Read Processor State Register
RDWIM	Read Window Invalid Mask Register
RDTBR	Read Trap Base Register
WRASR	Write Ancillary State Register
WRY	Write Y Register
WRPSR	Write Processor State Register
WRWIM	Write Window Invalid Mask Register
WRTBR	Write Trap Base Register
STBAR	Store Barrier
UNIMP	Unimplemented
FLUSH	Flush Instruction Memory
FPop	Floating-point Operate: FiTO(s,d,q), F(s,d,q)TOi, FsTOd, FsTOq, FdTOs, FdTOq, FqTOs, FqTOd, FMOVs, FNEGs, FABSs, FSQRT(s,d,q), FADD(s,d,q), FSUB(s,d,q), FMUL(s,d,q), FDIV(s,d,q), FsMULd, FdMULq, FCMP(s,d,q), FCMPE(s,d,q)

The rules for *SAVE* and *RESTORE* instructions are:

- On every *SAVE* instruction, the tag of the PC will be copied to the tag of the register window.
- For the *RESTORE* instruction, if the tag of the register window and the PC's tag are same or if Code-space(PC) > MANAGER, then the *RESTORE* is allowed. Otherwise the *RESTORE* is not allowed.

The read and write instructions for special purpose registers use the following rules:

- The *Y* register tag receives the tag of the written value. The other registers can only be accessed by non user code, but otherwise get the tag of written values.

The remaining instructions in this group are left for future work.

3.4.3 Concerns about the memory and stack in SPARC architecture

Assembly language implementation of the C programming language requires that we allocate memory to store local variables, functions, parameters, and linkage information in support of procedure calls. This memory region for each procedure is called a frame. Normally, these frames are allocated on a runtime stack, and pushed and popped with procedure calls and returns.

For the SPARC architecture, the frame contains space for registers, parameters, PC, and so on (Figure 21). From the assembly code in Figure 22, we can see that accesses to the frame normally use *%fp*, the frame pointer.

```
C code:
main()
{
   int a=25;
   int b=7;
   int c;
   c=foo(a,b);
}
int foo(int a1, int b1)
{
   int c1;
   c1=a1+b1;
   return c1;
}

Assembly code:
main:                                        ;main function
   save  %sp, -128, %sp              ;set the stack frame
   mov 25, %g1
   st  %g1, [%fp-20]                    ;int a=25;
   mov 7, %g1
   st  %g1, [%fp-16]                    ;int b=7;
   ld  [%fp-20], %o0                    ;put a in out register
   ld  [%fp-16], %o1                    ;put b in out register
   call  foo, 0                      ;call function foo
    nop
   mov %o0, %g1              ;move the returned value to %g1
   st  %g1, [%fp-12]         ;store the value to [%fp-12]
   restore                              ;restore caller's window
   jmp %o7+8                            ;jump back to the callers
    nop

foo:
   save  %sp, -112, %sp                 ;set the stack frame
   st  %i0, [%fp+68]            ;store a into [%fp+68]
   st  %i1, [%fp+72]            ;store b into [%fp+72]
   ld  [%fp+68], %g2            ;load a from memory space to %g2
   ld  [%fp+72], %g1            ;load b from memory space to %g1
   add %g2, %g1, %g1            ;add a and b, store in %g1
   st  %g1, [%fp-12]            ;store the result to [%fp-12]
   ld  [%fp-12], %g1
   mov %g1, %i0              ;move the result from %g1 to %o1
   restore                      ;restore caller's window
   jmp %o7+8                    ;jump back to the callers
    nop
```

Figure 22: Sample Assembly code 1

Stack memory is used to store frames, and the frames on the same stack in a ZKOS can belong to directive code or users. Therefore, we cannot treat frames the same as regular memory.

For example, after returns from a directive, the memory that has been used for that directive frame will be tagged with a tag containing Manager code-space. Therefore, if the user tries to call a new user function, we could not use the same memory for the user function frame without violating our assignment rules unless we treat the stack differently from regular memory. Therefore, stack memory is treated as if the copy-bit is set for all write access to the stack. Stack memory will therefore be limited to specific pages of memory.

In addition, we want to protect the calling function's frame from tampering by the called function. We have to restrict the use of the frame pointer or other pointers when accessing stacks. At this time, we only allow the frame pointer to access the contents of the stack.

To protect from using the frame pointer to overwrite memory, on every *SAVE* instruction that allocates a new stack frame in the main memory, we put the boundary of the stack frame in the tag of the frame pointer. Therefore, when using the frame pointer to access stack, the tag of the frame pointer will be checked to make sure the frame pointer is accessing the stack that is in the boundary of its stack. If the frame pointer is pointing to part of the stack which is out of the boundary, then the frame pointer is not allowed to read or write to that stack memory.

3.4.4 A Proposed Representation of Tags

In Section 3.3.5.4 we discussed a possible high-level representation of our tags which we used in the rest of this project. In this section, we will briefly describe a proposed low-level representation of the tags to show how we can map our system in to a 32-bit tag.

3.4.4.1 Owner field and Code-space field

We assume there will be a large number of combinations of the Owner field and the Code-space field. Therefore we determined to use the bits of the two fields of tag to show the relation between owners and code-spaces. We propose that each of the Owner and Code-space fields be represented with 12 bits, with the remaining 8 bits for the control-bits.

11 8	7 5	4 0
RTEMS Code (4 bits)	RTEMS Level (3 bits)	Component Id (5 bits)

Figure 23: Bit representation of Owner field and Code-space field

We use the higher 4 of the 12 bits to represent if it is system resource or not. If the higher 4 bits are 1111, this means that the data or code which has this tag is from the system, otherwise it is from users. For system code which is started with 1111, the lower eight bits are used to divide the code into smaller classes. Among the 8 lower bits, we use the higher 3 bits to indicate the level of the class (Shown in Table 16).

Table 16 : RTEMS levels (bits 5-7)

RTEMS Level	Bit representation
HIGH & Tag init	111
SCORE & SCORE internal init & SCORE init	110
SCORE internal internal	101
SCORE functions	100
Manager & Manager internal init & Manager init	011
Manager internal internal	010
Manager functions	001
Startup	000

Table 17: Using tags to represent security classes

Bits (12 bits)			Representation
RTEMS Level			
RTEMS / USER (4 bits)	RTEMS Level (3 bits)	Component ID (5 bits)	RTEMS Representation
1111 (RTEMS)	111	11111	Tag INIT, SCORE, HIGH
		00001-11110	SCORE private internal functions
		00000	
	110	11111	
		00001-11110	SCORE internal functions
		00000	
	101	11111	
		00001-11110	SCORE internal init functions
		00000	SCORE INIT
	100	11111	
		00001-11110	SCORE external functions
		00000	
	011	11111	MANAGER
		00001-11110	Manager private internal functions
		00000	
	010	11111	
		00001-11110	Manager internal functions
		00000	
	001	11111	
		00001-11110	Manager internal init functions
		00000	Manager INIT
	000	11111	
		00001-11110	Manager external functions
		00000	Startup
USER LEVEL			
1110 111 11111			USER
….			User1, User2, … UserN
0000 000 00000			LOW

Table 18: SCORE component ID

SCORE function	Component ID	SCORE function	Component ID
thread	00001	object	00010
initialization	00011	chain	00100
protected heap	00101	heap	00110
workspace	00111	error	01000
message	01001	ISR	01010
watch dog	01011	time	01100
user extension	01101		

Table 19: Manager component ID

Manager function	Component ID	Manager function	Component ID
initialization	00001	task	00010
semaphore	00011	manager	00100
interrupt	00101	barrier	00110
rate monotonic	00111	clock	01000
dual ported memory	01001	timer	01010
multiprocessing	01011	I/O	01100
user extensions	01101	event	01110
fatal error	01111	signal	10000
partition	10001	region	10010
tag	10011	user	10100

For the detailed bit representation of manager level functions and SCORE level functions, refer to Table 18 and Table 19.

A possible design is shown in Table 17. By doing this, we can easily identify the security class of a tag. For example, if the Owner field is 1111 100 00010 and the Code-space field is 1111 100 00010, then the security class of the code is {object, object}. If two Owner fields are given, such as 0010 000 10010 (User) and 1111 001 01101 (user extensions manager), then the least upper bound of these two classes is 1111 001 01101 (user extensions manager). The least upper bound of 1111 100 00110 (heap) and 1111 100 01001 (message) is 1111 110 11111 (SCORE).

3.4.4.2 Control-bit field

In addition to 12 bits for each of the User and Code-space fields, we set aside one control-bit for the copy-bit (discussed in Section 3.3.5.4).

To further protect the memory, we divided memory into three classes:

- **Stack Memory.** Stack memory is readable and writable, and is treated using register expression rules for stores, but not assignment rules.
- **Code Memory.** Code memory stores the executable and readable code. The “entry point” of a function has a special tag with it to indicate the correct place of executing a function.
- **Data Memory.** Data memory is readable and writable, and it is treated using assignment rules.

  We allocate 3 control-bits for memory type:

  - [1xx] - memory is readable and writable
  - [0xx] - memory is read-only
  - [x11] - memory is an entry point to an executable function
  - [x10] - memory is executable but not an entry point
  - [x01] - memory is stack memory and needs to be treated special
  - [x00] - memory is data memory

The possible values of the memory type are shown in Table 20. We use a world-readable bit (bit 3) to indicate that the tagged data can be read by all entities. This is used when the system (higher level) wants to give the user (lower level) permissions to access the data manipulated by the higher level, such as configuration data. We allocate another 3 bits for future use.

Table 20: Possible values of the memory type

Memory type	Read or read/write
Data memory (x00)	000 - read-only
	100 - read/write
Stack memory (x01)	001 - read-only
	101 - read/write
Code memory (executable) not entry point (x10)	010 - read-only
	110 - read/write
Code memory (executable) entry point (x11)	011 - read-only
	111 - read/write

3.4.5 Sample assembly code

Figure 22 shows simple C-code (lines 2-14), compiled with *sparc-rtems-gcc*, which is a cross compiler of C-code for RTEMS running on SPARC architecture, to generate assembly code (lines 17-45). Assume the *main* function is user code, and the *foo* function is directive code that has a tag (manager1, manager1) associate with it.

- Starting at line 18 in the program with the PC (user1, user1). Line 18 of the assembly code sets the stack frame for the *main()* function by allocating save space for all registers and space for the local variables *a, b, c*. Following the stack-frame conventions, the local variables will be stored "below" the frame pointer with *a* at location *[%fp-20], b* at *[%fp-16]* and *c* at *[%fp-12]*. Note that this convention assumes that *a* is at a lower location in memory than the other local variables, effectively as if it was pushed on the stack last. At this point we are treating the memory as a stack.
- Line 19 assigns a constant to *%g1*, giving *%g1* the tag of the PC (completing the right-hand side of line 4, evaluating the expression and giving the expression a tag).
- Line 20 then stores the value into memory location for a after completing the assignment tagging rules (completing the *a=25* from line 4).
- Lines 21 and 22 perform similarly for *b=7*.
- Lines 23 and 24 evaluate the parameters of the function call from line 7. The load checks ensure that the current thread is permitted to read the values and puts them in registers, in this case inheriting the tag of the parameters.
- Line 25 performs the function call using the function execution rules (completing the call of line 7). The new tag of PC is (user1, manager1).
- Line 34 allocates a stack frame on the stack for *foo* and allocates a new register window with the tag of (user1, manager1).
- Line 35 stores the parameter *a*, which is from *%i0* to the space *[%fp+68]* in the stack.
- Line 36 stores the parameter b to the space *[%fp+72]* in the stack. Note that both of the parameters passed to the *foo* function are from the in registers, which were the out register of the main function. Rules for store instructions are checked for the *ST* instruction on line 34 and 35 (completing the parameter pass).
- Then, on line 37 and 38, variables *a* and *b* are loaded from stack to *%g1* and *%g2*. The *LD* instruction ensures the parameters are allowed to be used by the current thread, which has tag (user1, manager1).
- Line 39 adds the value of *a* and *b*, then stores the result (c1) in *%g1*. This is an arithmetic instruction, which needs to follow the rule for arithmetic instruction and the rules for store instructions (completing the addition operation on line 12).
- Line 40 stores the result to stack.
- Line 41 and 42 store the result in *%i0*, which is the place for the return value.
- On line 43, RESTORE instruction restores the caller's window.

- Then, line 44 jumps to the address that is stored in *%o7+8*. For this jump, there is a check to make sure that the target address is tagged with the same tag of the previous PC (completing the return on line 13).
- Back to *main* function, the PC changes back to the previous value. Lines 27 and 28 store the returned value to *[%fp-12]* on the stack (completing the code on line 7).

3.5 EXTENSION AND EVALUATION OF THE TAGGING SCHEME

After designing the security tagging scheme, recent work has focused on implementation of the tagging scheme and proof of the security policies. Section 3.5 introduces the work that has been done so far and also proposes future work.

One of the goals of this project is to expand RTEMS to support multiple users. Therefore the concepts of normal users and superuser are proposed. The superuser has the abilities to create, delete and control the normal users while normal users do not have these abilities. The normal users are isolated from each other, but can talk to other users by using secured messages.

Although the security tagging scheme was proposed as part of earlier work, making it work was more complicated. To implement it, modifications of SPARC instruction simulator (SIS) are required to support tag checking and tag propagation. In addition, some tagging functions must be added to RTEMS to provide software support for tagging handling. Hooks are needed to be added in SIS and some of the instruction interpretations are required. For example, the normal LD instruction loads data from memory to registers. However, for the modified LD instruction, tags are checked first to make sure the current running thread can access the data, and then execute the instruction to load data from memory if the access is allowed according to the tagging rules. To ensure the tagging rules are proper and the modified interpretation of instructions are correct, 76 test cases are generated (consisting of more than 2000 smaller tests) which tested all of the modified instructions.

In addition to working on the implementation of tagging scheme, formal models of the tagging scheme are being developed. The formal models help showing that the UI Tagging scheme ensures some of the security policies. A simple tagging model has been set up using ACL2. Since in the lattice, security classes are arranged in a partial ordering, the proof is started by proving that the model has reflexive, transitive and antisymmetric security classes. In addition, the model for tagging scheme has been proved to be a lattice model. Future work will be focused on proving more complicated lemmas and theorems.

The remainder of Section 3.5 is organized as follows: Section 3.5.1 introduces the design of multiple user system and the tagging issues related to changing RTEMS. The test cases for testing the simulator and the tagging rules are explained in Section 3.5.2. Lastly, the future work that need to be done is discussed in Section 3.5.3

3.5.1 Support Multiple User System

Currently RTEMS is a single user multi-threaded model of execution. To make it a multi-user system, introduced the concept of “superuse”. A superuser is a user who is able to create, delete and control normal users. Normal users can only control themselves, but have no abilities to create, delete or control other users. The normal users are isolated from each other; however users in RTEMS are allowed to talk to other users.

The message manager in RTEMS is used to store messages which support communication in RTEMS. Message queues are used to hold the messages. When creating a message queue, RTEMS will generate a unique message queue ID for this message queue. Therefore in a multi-user system, if user1 wants to send messages to user2, it needs to create a message queue and let the superuser know that the messages are to be sent from user1 to user2 using message queue ID. When receiving messages, the system first goes to the superuser to check if the user has permission to use the specific message queue. To accomplish this, the superuser needs to maintain a table to record the permissions. When the message queue is deleted, the specific entry related to that message queue in the permission table maintained by superuser will be removed.

To have more control over users and tasks, a set of task IDs is provided to each user. This allows additional secure isolation between the tasks. To implement this, the user levels' owner field of the tag is divided into two parts - users and tasks. For the owner field of tags for superuser and different users, the higher 7 bits among the 12 bits are used to represent separate users and the lower 5 bits are used to indicate task IDs. For example, the 12 bits for the superuser is `1110 110 XXXXX`, where the `1110 110` indicates superuser and the `XXXXX` represents task IDs. Since only 7 bits are used to represent different users, it limits the number of users in the system to be user1 to user117; `0000 001 XXXXX` to `1110 101 XXXXX`. Including the superuser, there are 118 possible users in the system[11]. This does not influence the levels above the general user (USER) level, because these controls are only made on users and their tasks. The new bit representation for owner field of tags are shown in Table 21.

Table 21: New bit representation for owner field of USER and LOW levels

USER	`1110 111 11111`
Superuser	`1110 110 XXXXX`
User 117	`1110 101 XXXXX`
…	…
User 1	`0000 001 XXXXX`
LOW	`0000 000 00000`

To change RTEMS to a multi-user system, the system code needs to be modified to support multiple users. In addition, a user manager needs to be implemented in RTEMS to handle the superuser and its abilities to create, delete and control normal users. This is all part of the future work of this dissertation.

[11] For an embedded system, 118 users each with 32 possible subtasks should be sufficient.

3.5.2 TAGGING TEST CASES

To support the UI Tagging scheme, modifications on SIS are required to make it support tag checking and tag propagation when executing instructions. In order to test all the tag checking and propagations are working as intended, 76 test cases (around 68000 lines of C code) are generated and each of them contains up to 64 smaller tests.

Since at this time, RTEMS is not fully tagged, memory and the PC have to be tagged manually. In these test cases, the memory of the variables and the PC are manually tagged when the tag engine is off, then the test turns on tag engine and executes the statement. After that, the test turns off tag engine and prints the result of the tag propagation.

Table 22 shows the basic utility functions for dealing with tags and special bits. For example, tags are specified by using `make_tag(),` and then tag a word of memory with a specific tag using `_rtems_tag_word()`. To check the tag of a memory, the function `_rtems_get_memory_tag()` can be used to return the tag of a specific memory. The PC is tagged and the tag of the PC can be checked with `_rtems_tag_pc()` and `_rtems_get_pc_tag().` In addition, domination relation of two labels (label represents a 12 bits owner field or code-space field of a tag) or security classes of two tags can be checked. Function `tag_lub()` is used to calculate the least upper bound of two tags. What is more, there exist functions that set or unset the copy bit and world readable bit, and change the memory types for testing purpose only.

Table 22: Summary of Test Case Utility Functions

Function Name	Function Description
make_tag()	Make a 32-bit tag of three fields. (Owner, Codespace, Control)
set_copy_bit()	Set the copy bit of a tag
reset_copy_bit()	Unset the copy bit of a tag
_rtems_tag_pc()	Tag the PC (Program Counter) with a specific tag
_rtems_get_pc_tag()	Get the tag of the PC
_rtems_tag_word()	Tag the word with a specific tag
_rtems_get_memory_tag()	Get the tag of a memory
label_lub()	Calculate the LUB of two labels
label_glb()	Calculate the GLB of two labels
label_dominates_or_eq()	Give the domination relationship of two labels
tag_lub()	Calculate the LUB of two tags
tag_glb()	Calculate the GLB of two tags
tag_dominates_or_eq()	Give the domination relationship of two tags
set_world_readable()	Set the world readable bit of a tag
reset_world_readable()	Unset the world readable bit of a tag
set_read_write()	Set the memory type to read/write
set_read_only()	Set the memory type to read-only
set_code_mem_entry()	Set the memory type to an entry point of a code memory
set_code_mem_not_entry()	Set the memory type to a non-entry point of a code memory
set_stack_memory()	Set the memory type to stack memory
set_data_memory()	Set the memory type to data memory

To make the test cases easier to understand, two sample tests are provided here. Source code and outputs are provided in Figure 24 and Figure 25.

The first test case (test 4-1) evaluates what the system does when copying from variable `value2` to variable `value1` when neither has the copy bit set and the security level of the program counter exceeds the security level of `value2` which is greater than the level of `value1`; this should not throw an exception. Information about the test case is provided on lines 11-12 of Figure 24, and corresponding output on lines 1-2 of Figure 25.

- On lines 5 to 8, four tags are created. tag4_ucp stores the tag (USER1, WATCHDOG_EXT, false, READWRITE|DATAMEMORY). The owner field of the tag is USER1, code-space field is WATCHDOG_EXT (watchdog external function). The control field shows the tagged data is read/write data memory with the copy bit not set. The tag4 stores a tag that similar to tag4_ucp, but has the copy bit set. tag2_ucp stores tag that the code-space field is REGION_EXT (region external function). tag1_ucp has tag that both owner field and code-space field are USER1, and copy bit not set.

- On line 10, a CPOP_DEBUG_ON instruction is used to turn on the debugging. After turning on the debugging, error information will be printed when a tagging exception generated.

- From line 11 to 26 is one of the 42 small tests (test 4-1) in test 4 and lines 29 to 43 are for another small test (test 4-32) from test 4. The outputs of these tests are shown in Figure 24.

- On line 13, there are two variables: value1 and value2. They are initialized to 1 and 10.

- On lines 15 to 16, _rtems_tag_word() is used to give tags to value1 and value2. After executing line 15 and line 16, value1 will be tagged with (USER1, REGION_EXT, false, READWRITE|DATAMEMORY), and value2 will be associated with tag (USER1, USER1, false, READWRITE|DATAMEMORY).

- On line 17, _rtems_tag_pc() function gives the PC a tag (USER1, WATCHDOG_EXT, false, READWRITE$\mid$DATAMEMORY).

- Before turning on tagging scheme, lines 18-20 print the tags or value1, value2 and PC.

- On line 22, the tagging scheme is turned on. The function start_tagging() includes telling GCC to not use all the registers\footnote{There may be unexpected interruptions if GCC uses the values stored in the registers.}, and then executing CPOP_TURN_ON_TAGGING instruction (asm (CPOP_TURN_ON_TAGGING);) to turn on tagging scheme.

- Line 23 performs the assignment value1 = value2;.

```
void test4()
{   tag_t tag4, tag1_ucp, tag2_ucp, new_tag, tag4_ucp;
 char str[300];
 volatile uint32 value1, value2;
 tag4_ucp = make_tag(USER1,WATCHDOG_EXT,false,READWRITE|DATA_MEMORY);
 tag2_ucp = make_tag(USER1,REGION_EXT,false,READWRITE|DATA_MEMORY);
 tag4 = make_tag(USER1,WATCHDOG_EXT,true,READWRITE|DATA_MEMORY);
 tag1_ucp = make_tag(USER1,USER1,false,READWRITE|DATA_MEMORY);

 asm(CPOP_DEBUG_ON);
 printf("\n-value1 cp not set, value2 cp not set-\n");
 printf("****** 4-1. value2 < value1 < PC ******\n");
 value1 = 1; value2 = 10;
 printf("value1 = %d, value2 = %d\n",value1, value2);
 _rtems_tag_word((addr_t)&value1, tag2_ucp);
 _rtems_tag_word((addr_t)&value2, tag1_ucp);
 _rtems_tag_pc(tag4_ucp);
 printf("PC tag: %s\n", tag_to_string(str,_rtems_get_pc_tag()));
 printf("value1 tag: %s\n", tag_to_string(str,
     _rtems_get_memory_tag((addr_t)&value1)));
 printf("value2 tag: %s\n\n", tag_to_string(str,
          _rtems_get_memory_tag((addr_t)&value2)));

 start_tagging();
 value1 = value2;
 asm(CPOP_TURN_OFF_TAGGING);
 new_tag = _rtems_get_memory_tag((addr_t)&value1);
 printf("OPERATION: value1 = value2, new tag of value1(%d) is:\n %s\n",
            value1, tag_to_string(str, new_tag));
 ...
 printf("\n-value1 cp set, value2 cp not set-\n");
 printf("****** 4-32. value1 < pc < value2 ******\n");
 value1 = 32; value2 = 320;
 printf("value1 = %d, value2 = %d\n",value1, value2);
 _rtems_tag_word((addr_t)&value1, tag1_ucp);
 _rtems_tag_word((addr_t)&value2, tag4);
 _rtems_tag_pc(tag2_ucp);
 printf("pc tag is: %s\n", tag_to_string(str,_rtems_get_pc_tag()));
 printf("value1 tag: %s\n", tag_to_string(str,
          _rtems_get_memory_tag((addr_t)&value1)));
 printf("value2 tag: %s\n\n", tag_to_string(str,
          _rtems_get_memory_tag((addr_t)&value2)));

 start_tagging();
 value1 = value2;
 asm(CPOP_TURN_OFF_TAGGING);
 new_tag = _rtems_get_memory_tag((addr_t)&value1);
 printf("OPERATION: value1 = value2, new tag of value1 (%d) is:\n %s\n",
          value1, tag_to_string(str, new_tag));
 ...
 asm(CPOP_DEBUG_OFF);
```

Figure 24: Sample test case

- On line 24, the tagging scheme is turned off by using asm(CPOP_TURN_OFF_TAGGING).

- After executing the assignment while having the tagging scheme turned on, the tag should be propagated correctly as specified by the tagging rules.

- On line 25, function _rtems_get_memory_tag((addr_t) & value1) is used to get the new tag associated with value1. According to the tagging rules for assignment (See Section 3.3.6.4) if both value1 and value2 have copy bit not set, to allow the information flow form value 2 to value1, the security level of value1 and value2 have to be lower than the level of PC's security level. On line 17, PC is tagged with tag (USER1, WATCHDOG_EXT, false, READWRITE|DATAMEMORY). Since value1 has tag (USER1, REGION_EXT, false, READWRITE|DATAMEMORY) and value2 is tagged (USER1, USER1, false, READWRITE|DATAMEMORY), both security levels of value1 and value2's tags are lower than the security level of PC'tag. Therefore the assignment is allowed and no exception will be generated. The value of value2 will be stored in value1, but the tag of value1 should not be changed based on the tagging rules.

- On line 26, the value of value1 will be printed as well as the tag of it.

The second listed test case (test 4-32) evaluates what the system does when copying from variable `value2` to variable `value1` when the copy bit is set for variable `value1` and the security level of `value2` exceeds the security level of the program counter which is greater than the level of `value1`; this should not throw an exception. Information about the test case is provided on lines 28-29 of Figure 24, and corresponding output on lines 11-12 of Figure 25).

- Lines 32 to 34 show that in this test (test 4-32), value1 is tagged (USER1, USER1, false, READWRITE|DATAMEMORY), value2 is tagged (USER1, WATCHDOG_EXT, true, READWRITE|DATAMEMORY) and PC has tag (USER1, REGION_EXT, false, READWRITE|DATAMEMORY).

- On lines 39 to 41, the tagging scheme is turned on and the assignment is executed. However, according to the tagging rules (if the copy bit of value1's tag is $\overline{cp}$ and the copy bit of value2's tag is *cp*, to allow the assignment, the condition $\underline{value1} \leq \underline{PC}$ and *Owner(value1) = Owner(value2)* have to be met.) Because both the owners of value1 and value2's tags are USER1, and the security level of the tag of value1 ({USER1, USER1}) is at lower level than the security level of PC's tag ({USER1, REGION_EXT }) , the assignment is allowed. The tag of value2 will be copied to the tag of value1.

Figure 25 shows the output of the sample tests. Lines 1 to 9 are output for test 4-1 and lines 11 to 19 are output for test 4-32. For test 4-1, when both copy bits of value1 and value2's tags are not set, after executing the assignment (value1 = value2;), value1 gets value2's value but its tag doesn't change (see line 8 in Figure 25). For test 4-32, the copy bit of value2's tag is *cp* while $\overline{cp}$ for value1's tag. Since the copy bit of value2's tag is set, value2's tag will be copied to value1's tag (see line 19 in Figure 25).

```
 1   -value1 cp not set, value2 cp not set-
 2   ****** 4-1. value2 < value1 < PC ******
 3   value1 = 1, value2 = 10
 4   PC tag: <User 1,Watchdog External,CP=false,WorldReadable=false,Read-Write Data Memory>
 5   value1 tag: <User 1,Region External,CP=false,WorldReadable=false,Read-Write Data Memory>
 6   value2 tag: <User 1,User 1,CP=false,WorldReadable=false,Read-Write Data Memory>
 7
 8   OPERATION: value1 = value2, new tag of value1(10) is:
 9    <User 1,Region External,CP=false,WorldReadable=false,Read-Write Data Memory>
10
11   -value1 cp set, value2 cp not set-
12   ****** 4-32. value1 < pc < value2 ******
13   value1 = 32, value2 = 320
14   pc tag is: <User 1,Region External,CP=false,WorldReadable=false,Read-Write Data Memory>
15   value1 tag: <User 1,User 1,CP=false,WorldReadable=false,Read-Write Data Memory>
16   value2 tag: <User 1,Watchdog External,CP=true,WorldReadable=false,Read-Write Data Memory>
17
18   OPERATION: value1 = value2, new tag of value1 (320) is:
19    <User 1,Watchdog External,CP=true,WorldReadable=false,Read-Write Data Memory>
```

Figure 25: Sample test case output without exception

In another test case (test 4-33), PC has tag (USER1, USER1, false, READWRITE|DATAMEMORY). Value1 is tagged (USER1, REGION_EXT, false, READWRITE|DATAMEMORY) and value2 is tagged (USER1, WATCHDOG_EXT, true, READWRITE|DATAMEMORY). Similar to test 4-32, the copy bit of value2's tag is *cp* while $\overline{cp}$ for value1's tag. However, the assignment will not be allowed, because the condition $\underline{value1} \leq \underline{PC}$ is not met. Therefore, a security exception will be generated. A tagging exception handler will handle this and print out information about the error. As shown in Figure 26, lines 7 to 14 provide information about the violation. It indicates that the violation is caused by `ST` instruction and the reason is the relationship of PC's tag and value's tag is not satisfied.

Because of the violation, the assignment will not be executed. Therefore on lines 15 and 16 (Figure 26) the new tag of value1 is its original tag and the value of `value2` (330) is not copied to `value1`.

The future work of testing tagging schemes includes testing the support of multiple users. Because the current RTEMS will be modified to support multiple users, handling tags for superuser and different normal users would be an additional work.

Approved for Public Release; Distribution Unlimited.

```
****** 4-33. pc < value1 < value2 ******
value1 = 33, value2 = 330
pc tag is: <User 1,User 1,CP=false,WorldReadable=false,Read-Write Data Memory>
value1 tag: <User 1,Region External,CP=false,WorldReadable=false,Read-Write Data Memory>
value2 tag: <User 1,Watchdog External,CP=true,WorldReadable=false,Read-Write Data Memory>

ST_IMM2 CHECK ERROR when copy bit of source is set and addr is not set, non-stack memory
tag_pc = <User 1,User 1,CP=false,WorldReadable=false,Read-Write Data Memory> does NOT
DOMINATE
tag_addr = <User 1,Region External,CP=false,WorldReadable=false,Read-Write Data Memory>

...
Hit vector 0x28!! @ PC = 0x 20EE484 with NPC = 0x20EE488
RETT at 0x20fb900 with addr =0x20ee48c
Violating Instruction: 0xC227BFFC :: ST
OPERATION: value1 = value2, new tag of value1 (33) is:
<User 1,Region External,CP=false,WorldReadable=false,Read-Write Data Memory>
```

Figure 26: Sample test case output with exception

3.5.3 Future Work

This section summarizes proposed future work for this project.

Fully tag RTEMS code to enables RTEMS to always use tagging: Currently, all of the test cases work by enabling and disabling tagging as needed. Therefore only some of the RTEMS code is tagged. For the future work, all of the RTEMS functions and C library functions that are used by the test cases have to be tagged. The goal is to have a tagged RTEMS running from the beginning of the system initialization. To evaluate the system, portions of RTEMS and C libraries that are needed by the benchmark applications are going to be tagged. This step may be time consuming, because RTEMS requires a fresh build and installation even with a small change of the code, which takes around 20 minutes. Documentation of the conversion process will be created to allow future more complete conversion of the code and libraries.

Modify RTEMS code to support multiple users: Modification of RTEMS source is needed to support multiple users, especially the use of global variables. Some global variables can be shared among different users, because those global variables will not affect the security of the isolated users and the whole system as well. However, some global variables may affect other users or system will need to be made usable only by a specific user.

In addition to global variables, the C library functions need to be considered as well. Because of the time issue, not all of the C library functions that are used by RTEMS can be gone through manually. Therefore software tools will be used to generate a list of the C library functions that are used in the RTEMS benchmarks. These functions will be given special tags. By doing this, additional checks will be applied when using these functions, to prevent malicious usage of the important functions.

Implement user manager to support multiple users: A user manager will be added in RTEMS to deal with the tagging support for multiple users. For example, the user manager should be able to check whether it being called by a superuser or a normal user, should be able to create and delete a normal user, and it should be able to maintain a table which record the current users.

Because only the superuser has permissions to create, delete and control other users, it requires the user manager to figure out that the current user is superuser or normal user. What is more, the manager needs to maintain a table to record all of the current users in the system. According to this table, the manager should be able to allocate a proper tag when creating a new user.

Modify the SIS to make it support multiple users: Currently, hooks and additional instruction interpretations are successfully added in SIS for the initial tagging scheme. However, since the UI Tagging scheme will be expanded to support a multiple users system, additional changed in the SIS will be required, such as applying advanced tag checking and propagation rules to SIS. This future work will be done mostly by other team members using the model developed in this dissertation.

Develop test cases to test the correctness of the modified RTEMS and SIS: After changing the RTEMS and SIS to support multiple users, new test cases are need to be generated to ensure the correctness of the implementation. Since the initial implementation of tag checking and propagation have been tested, only the support of multiple users are need to be tested and evaluated.

Port sample applications to evaluate the performance of the tagging scheme: In addition to the simple test cases, some sample applications will be ported to RTEMS. These benchmark applications will help evaluate the performance of the UI Tagging scheme. This may require additional changes in RTEMS code, for RTEMS is only compatible with a few applications and none of them are usable for evaluation purpose. Based on the evaluation results, other methods, such as improving the performance, may be proposed. For example, tag compression can be implemented to reduce the time that used for handle tags. This porting work will be done by other team members, and the applications will be used to assist in the evaluations of this dissertation.

Proofs of correctness and security: A simple formal security policy model of the tagging system has been set up, and the system is a lattice can be proved. In the future, more proofs of the correctness of the tagging implementation will be proved as well as the formal policy to validate that the implemented system can support higher-level system security properties.

In the future, the simple formal model of security policy will be expanded to a model that represents multiple users. More lemmas will be proven using that advanced formal model. The proofs may focus on proving the advanced model is a Multiple independent levels of security (MILS) like model which ensures the separation and controlled information flow within the model [Alves-Foss06].

3.6 TAGGING AND SECURITY POLICIES

There are several different definitions for a computer security policy, but in general a computer security policy defines the set of resources of a system and the authorized operations on those resources. In the high assurance community, security policies often take one of two forms: access control policies and information flow policies.

3.6.1 Access Control Policies

An access control policy specifies when a subject (a user or a process) can perform an action on an object (a resource of the system). That action can be as simple as a read or write operation to a more complex business or control system transaction. Access control policies require a specification of system resources and users, an assignment of security labels to those entities, and relationship operations between different security labels. In addition, we need assurance that the mechanisms that enforce the policy satisfy the following constraints (these are based on the properties first set out in the Anderson report [Anderson72]:

- *Always invoked*: Upon every operation, there is a check for access control permissions. This guarantees that any changes to the permissions or security labels is enforced at the time of action and access is not granted based on a legacy decision. For example, an access control system for a file system may use metadata stored on the disk drive to make an access decision. If a user requests to open a file, the system may check the metadata and then cache the result of the check. Upon future requests, if the system checks the cached result, it may miss changes to the metadata and therefore violate the always invoked property.
- *Non-bypassable*: There is not a way to bypass the access control mechanisms and directly access the resource. Consider the file system example above. If the user is able to directly read data from the disk drive they can bypass the file system and associate access controls.
- *Tamperproof*: The security system and metadata must be protected from unauthorized modification. Using the file system example, if a user can modify the metadata, or the code that evaluates the checks, then the system will not be secure.
- *Evaluatable*: We need assurance that the system performs as specified. That can only be done if the security mechanisms are isolatable from the non-security relevant portions of the code. A secure system must be structured and designed so that it can be reviewed and provide a high level of confidence in the correctness of the system.

According to Schneider [Schneider00] we can implement access control security mechanisms using run-time enforcement monitors. These monitors are non-bypassable and always invoked, and they check the behavior of the monitored system (e.g., in the file system example they monitor the user file access requests). If an attempted security violation is discovered, they either terminate the offending processes, suppress the behavior, or modify the behavior (e.g., change the request, insert new actions, etc.) [Ligatti05]. Such run-time enforcement behavior requires a system that satisfies the preceding constraints.

We have reviewed several hardware tagging schemes (see Section 3.1), in light of these constraints and have come to the following conclusions.

3.6.1.1 Limitations of Tagging

Most of the hardware based tagging schemes presented in the literature are limited to protection against buffer overflow attack (bounds checking violations), uninitialized memory usage, or other run-time violations of the code. Specifically, data is tagged with a simple indication of provenance:

- *User data:* Data that comes from an external source is considered tainted by these schemes and is marked by software (typically operating system drivers) that brings that data into the system. Hardware protections are in place to prevent the use of this tainted data in control flow operations; and to propagate the taint tags.
- *Internal data tagging:* In addition to user data, some of the schemes tag data with a *color* to indicate membership in a data group (usually adjacent buffers are differently colored) or to indicate the memory region is uninitialized. Hardware protections are in place to prevent overflow between differently colored regions or use of uninitialized memory regions.
- *Fat pointers:* Some of the tagging schemes add additional information to a pointer, indicating base and bounds of the region being reference. Hardware protections are put in place to ensure that the region access by a reference pointer is not out of bounds.

More complex tagging schemes, such as the one we presented in this report, add security labels or data types in the tags. These labels provide a richer set of control over data access, information flow and flow control.

Hardware based tagging schemes have the following limitations:

Management of tags: Any tagging scheme, beyond the basic DIFT approach, requires tag management functions. This includes: initial tagging of memory and possible registers, starting and stopping of the tag co-processor (to allow for initialization and error handling), exception processing including retrieving the tags used to generate the error, setting or changing the tags, and access controls to the tagging functions. More complex tagging may involve user tag definition and therefore strong access controls to the tagging functions.

Memory constraints: Most proposed tagging schemes have large amounts of tag overhead, because the hardware does not have organizational knowledge of the data structures and tag domains in the programming language and therefore must be conservative and provide more tags than necessary. A software-based approach can use compile-time and run-time knowledge of data structures within a hierarchical memory organization to provide implicit tags to the data.

Semantic restrictions: The hardware designer, and therefore the tag co-processor have limited semantic knowledge of the run-time tagging needs of the software. A tagging scheme provides basic labelling capabilities with tag propagation and checks based on a fixed (or limited) set of rules, and cannot provide direct enforcement of higher level security policies.

Approved for Public Release; Distribution Unlimited.

3.7 APPLICATION TAGGING

There have been claims in the security tag architecture community that hardware tags can now be used to address higher level attacks; those that are specifically attacking application logic, and not fundamental flaws in the underlying operating system, services or programming language. We call these types of attacks, *semantics attacks*, in that they result from a semantic disconnect between the programmer's view of interactions with the world, and the attackers actions. In Section 3.7 we provide a brief overview of these types of attacks and then discuss the use of a STA security mechanism to help prevent them.

3.7.1 Application Level Vulnerabilities

In October 2000, Bruce Schneier wrote a newsletter article for "Crypto-Gram Newsletter" where he provided commentary on different levels of network/security attacks [Schneier00].

In this article he wrote,

"The first wave of attacks was physical: attacks against the computers, wires, and electronics. Over the past several decades, computer security has focused around syntactic attacks: attacks against the operating logic of computers and networks. This second wave of attacks targets vulnerabilities in software products, problems with cryptographic algorithms and protocols, and denial-of-service vulnerabilities -- pretty much every security alert from the past decade. The third wave of network attacks is semantic attacks: attacks that target the way we, as humans, assign meaning to content."

The levels of attack that Schneier described were paraphrased from the work of Libicki [Libicki95]. In his article Schneier simply stated,

> "People are already taking advantage of others' naivete. ... Computer networks make it easier to start attacks and speed their dissemination, or for one anonymous individual to reach vast numbers of people at virtually no cost. ... semantic attacks will be more serious than physical or even syntactic attacks, because semantic attacks directly target the human/computer interface, the most insecure interface on the Internet" [Schneier00].

Typically, an individual (user) accesses an Internet web page to conduct a transaction. During this transaction, the user will be required to provide sensitive information, such as a credit card number. The website where the user provides this information should be a secure web server, which is nothing more than a computer that uses encryption to safely communicate and store sensitive information. Since the merchant cannot access credit card information directly, the merchant will most likely use a payment gateway. This payment gateway has the responsibility of determining the credit card company, and then contacting it via the credit card company's secure server. The credit card company will examine the transaction against stored account information, and the credit card company will either authorize or deny the transaction. The credit card company will pass this information back through the payment gateway. The payment gateway will then provide an acceptance or a denial back to the merchant's secure web server. This whole web transaction is considered secure if the information within the system cannot be accessed and/or modified by unauthorized users [NIST:800-47].

Web application vulnerabilities are tracked by The Open Web Application Security Project (OWASP) [OWASP10]. OWASP's mission is to make security visible, and to educate individuals about vulnerabilities and defenses. OWASP maintains a top ten list of vulnerabilities, where a vulnerability is a weakness or flaw in the application that allows an attacker to cause harm. According to OWASP, the top ten vulnerabilities [OWASP10] are:

1. *Injection* - Injection occurs when untrusted data is sent to the target interpreter and that untrusted data allows for an exploit in the interpreter. This exploit can result in data loss, data corruption or even complete system takeover. Injection flaws are very prevalent, particularly in legacy code, such as code found in SQL queries, operating system (OS) commands, and many other systems.

2. *Cross-Site Scripting (XSS)* - XSS is currently the most prevalent web application security flaw. XSS flaws occur when an application sends untrusted content or non-sanitized content to a web browser. XSS flaws allow the attacker to execute selected code on the victim's browser. There are three known types of XSS flaws: 1) Stored, 2) Reflected, and 3) Document Object Model (DOM) based XSS.

   - Stored attacks are those where injected code is permanently stored on target servers; such as in a database. The victim retrieves the malicious code from the server when it requests stored information.

   - Reflected attacks occur when code is injected into the web server through means such as an error message. Reflected attacks are typically delivered to a victim via another route, such as in an e-mail message, where the user is tricked into clicking on a malicious link. The injected code travels to the vulnerable web server, which reflects the attack back to the victim's browser. The browser executes what appears to be trusted code; however, the code being executed is malicious.

   - DOM XSS occur when the attack is executed as a result of modifying the DOM in the user's browser. This will cause the client side code to run in an unexpected manner. The page the user sees does not change, but the client side code contained in the page executes differently due to the attack.

3. *Broken Authentication and Session Management* - User authentication is common for many web applications. Broken Authentication and Session Management attacks occur when authentication and session management schemes do not adequately protect user credentials for all aspects of the site. The flaws in these schemes frequently cause problems with logout, password management, timeouts, and account updates.

4. *Insecure Direct Object References* - Applications frequently use the actual name of an object when generating web pages. Applications that don't verify the user, could allow an insecure direct object reference flaw.

5. *Cross-Site Request Forgery (CSRF)* - CSRF is an attack that forces an authenticated user to execute unwanted actions on a web application. When browsers send credentials, such as session cookies, attackers create malicious web pages which generate malicious requests that are indistinguishable from legitimate ones. A successful CSRF attack can compromise the user's data, such as changing the victim's password or email address.

6. *Security Misconfiguration* - Security misconfiguration occurs when the attacker gains unauthorized access because the system was not configured properly. Attacks can occur via access to default accounts, or by exploiting unpatched vulnerabilities. Security misconfiguration can happen at all levels of the application process, including the web server, application server, and code framework.

7. *Insecure Cryptographic Storage* - Insecure Cryptographic Storage occurs when data is not encrypted when it should be. If encryption is used, and the key generation is unsafe, then it is easy to determine the encryption key. Insecure Cryptographic Storage failure frequently compromises data that should have been encrypted.

8. *Failure to Restrict URL Access* - Failure to Restrict Uniform Resource Locator (URL) Access occurs when web pages do not properly protect page requests; commonly caused by incorrect page configurations. An attacker can gain access to the system by changing the URL to a privileged page URL. This could allow the attacker to access privileged account information.

9. *Insufficient Transport Layer Protection* - Insufficient Transport Layer Protection occurs when web applications do not protect network traffic, by not correctly requiring authentication. Without authentication, data and session variables can be exposed.

10. *Un-validated Redirects and Forwards* - Unvalidated Redirects and Forwards occur when an application redirects users to other pages, without validating the target page. This type of flaw can allow the attacker to force the victim's browser to open a specified, possibly malicious, web page.

In order to remove the chance of vulnerabilities, it is up to the developer to understand them and then properly code to protect against them. Unfortunately, expecting the developer to code against vulnerabilities is also problematic. The developer may not know that vulnerabilities even exist; or worse yet, the developer knows there are vulnerabilities, but chooses to ignore them. In either case some other mechanism needs to be pursued to secure the web application from

Approved for Public Release; Distribution Unlimited.

vulnerabilities. One such mechanism is runtime enforcement which is explained in the following section.

3.7.2 Semantic Attack Definition/Code Injection Attacks Definition

The precise definition of a *semantic attack* varies greatly. In Schneier's original article when he referred to semantic attacks he was referring to the "human element" and attacks such as modern day phishing or pump-and-dump schemes. This type of definition has become the definition of "semantic hacking." According to PC Magazine one definition of semantic attack is: "The use of incorrect information to damage the credibility of target resources or to cause direct or indirect harm. Examples include defamation (slander and libel), propaganda and stock manipulation schemes. Also known as 'semantic hacking.' " [PCMag]

Other definitions of semantic attacks refer to some type of malicious code injection into the web browser or the running code. For this dissertation, a semantic attack will be defined as an attack where malicious code is inserted into a running program. This definition of semantic attack classifies the semantic disconnect between the application programmer's mental model of the external environment and the actual reality of the operational environment. In most cases an injection attack will occur based on the application programmer not correctly validating user supplied input.

If one examines the OWASP top ten vulnerabilities, it is evident that most of those vulnerabilities are considered semantic attacks; yet, those attacks are also classified as injection attacks, which refers to part of the attack process.

3.7.2.1 SQL Injection

OWASP classifies injection attacks as the number one attack for web applications, in particular SQL Injection. SQL injection occurs when an application builds an SQL query using user input that is not appropriately filters and then sends that query to the interpreter.

For example, the normal query might appear as:

```
SELECT * FROM user_table
    WHERE email = `User email supplied from a web form';
```

Assuming the user entered in the web form: `name@address.com` the query string would be expanded to:

```
SELECT * FROM user_table
    WHERE email = `name@address.com';
```

It is important to understand that the single quotes that surround the email address and the ending semicolon are important aspects of the SQL query. The semicolon indicates the end of the query string. The single quotes indicate the beginning and end of the user input. Since the interpreter is building a string, it searches for the opening and closing single quotes. If there is not an equal number of quotes (too many or too few) then the interpreter will abort with a syntax error, and an error message will be displayed to the user. In the above example, the single quote marks align. Therefore, as an unfiltered query this query would return the desired results.

Unfortunately, the above query can easily be altered to add (or inject) additional SQL commands into the query. Instead of entering the email address, assume that the user enters the following:

```
name@address.com' or '1' = '1'; --
```

In SQL the double dash indicates a comment so everything after the double dashes is ignored, which block additional SQL constructs from the web page. Expanding out the new query string, the command would appear as

```
SELECT * FROM user_table
    WHERE email = `name@address.com' or ‘1’ = ‘1’; -- ‘;
```

Since “`or ‘1’ = ‘1’`” is always true, then the interpreter would return every item in the database. In this very simplified example we still return ever entry from the user table. More complex injection attacks can lead to modification or destruction of the database.

3.7.2.2 SQL Injection Security Mechanisms

A SQL injection is very difficult to detect because, to the database, the SQL injection appears as a normal SQL query from an authorized client (the web server). There are three main security mechanisms that protect against SQL injection.

- **Application Developer:** Most security policies to combat SQL Injection are implemented on the application developer's end. It is the responsibility of the application developer to filter the user input. This is typically accomplished through language specific filtering functions. For example, in the PHP language there is a function called mysql_real_escape_string(). This function prepends backslashes to a set of characters, including the single quote. The problem with leaving it to the application developer is, the developer may not know or even understand SQL injection.

- **Filtering and Monitoring Software:** Filtering and monitoring tools at the Web application and database levels will help block attacks and detect attack behavior. At the application level, organizations can possibly prevent SQL injection by implementing runtime security monitoring; furthermore web application firewalls can help organizations by creating behavior-based rule sets to block SQL injection attacks. Database activity monitoring can filter attacks on the back end, especially for known SQL injection attacks. There are generic filters that query for typical SQL injections such as uneven numbers of quotes. The drawback with filtering and monitoring software is two-fold. The software can be very expensive, which often times a small business can't afford, and if the business is able to afford the software, the installation/setup is often beyond the database/business owner's skill set.

- **Database Patches:** The risks associated with SQL injections are increased when the databases tied to the web applications are poorly maintained, including poor patching and configuration. Part of the configuration process requires better management on web application's associated accounts, especially with accounts that interact with the back-end databases. Many problems arise due to database administrators not understanding security, so the administrators give the web application accounts greater privileges than required. These super accounts are very vulnerable to attack and thus greatly broaden the risks to databases.

3.7.2.3 Cross-site Scripting

Cross-site Scripting flaws occur when an application includes user-supplied data in a page sent to the browser without properly validating or "escaping" that content. Cross-site Scripting vulnerabilities target scripts embedded in a page which are executed on the client-side, in the victim's web browser, rather than on the server-side, where the web page is held. Cross-site Scripting, in itself, is a threat which is brought about by the security weaknesses of client-side languages such as HTML, PHP and JavaScript. The concept of Cross-site Scripting is to manipulate client-side scripts of a web application to execute in the manner desired by the malicious user. Such a manipulation can embed a script in a page which can be executed every time the page is loaded, or whenever an associated event is performed. Cross-site Scripting vulnerabilities can have significant consequences such as tampering and sensitive data theft.

Essentially, Cross-site Scripting allows the attacker to execute selected code on the victim's browser. Once the page is loaded the victim's machine is compromised. A comprised machine could:

- Allow for the victim's cookies, which often contain username and password to be stolen
- Control the browser remotely
- Spread worms

A simple example:

The URL on the site `http://www.mysite.com/search?q=plants` returns HTML containing

```
<p> Your search for 'plants' returned the following results: <p>
```

The value of the query parameter `q` (in the URL) is inserted into the page returned by the site. Since the data is not validated, filtered or escaped (sanitized) then a malicious attacker could put up a page that causes a malicious URL to be loaded in the browser.

```
http://www.mysite.com/search?q=flowers+\%3Cscript\%3EevilScript()\%3
C/script\%3E
```

When a victim's browser loads the URL above. The document loaded will contain:

```
<p> Your search for 'plants <script> evilScript() </script> ' </p>
```

Loading this page will cause the browser to execute `evilScript(),` which might allow the browser to be controlled remotely.

3.7.2.4 Cross-site Scripting Security Mechanisms

Like SQL Injection, Cross-site Scripting is also very hard to detect. There are three main security mechanisms that protect against Cross-site Scripting.

- **Filtering and Monitoring Software**: Similar to SQL Injection filtering and monitoring tools can be applied at the Web application level. As with SQL injection there are the two drawbacks of expense and insufficient skills.

- **Validate input**: Validating input is very complex. A good programmer will validate the input received; however, a novice programmer has no idea about Cross-site Scripting and how to validate the input. Validating input quickly fails when the programmer has no knowledge of the problem.

- **Allow the client to disable client-side scripts:** In order for Cross-site scripting to be effective the malicious script needs to run in the victim's browser. By disabling scripts the client browser cannot be compromised. This is an effective mechanism; however, it relies on web sites not requiring or needing scripts to display the web page. Without some kind of scripting, the web pages would not be dynamic. In our modern world, most users rely on dynamic web content to enhance their web experience.

3.7.3 Summary of Semantic Attack Security Mechanisms

The above sections outline the two most prevalent web vulnerabilities and the security mechanisms required to prevent them. In every case, the required security mechanism relies on the programmer to implement the mechanism, and then handle any security violations. First, it presumes the programmer fully understands what security mechanism is required. If the programmer knows the required security mechanism, then it presumes the programmer knows how to properly code that mechanism. Other defenses are also problematic. Filtering and monitoring are often prohibitive based on the cost or the system administrator knowledge.

3.7.4 Hardware Tagging for Semantic Attack Prevention

There have been claims in the literature that some hardware tagging schemes can assist in the prevention of semantic attacks, such as SQL-injection and cross-site scripting [Dalton07, Kannan09]. The concept is understandable; this is user supplied data that is being processed in an unexpected manner, resulting in a violation.

As mentioned in the previous section, prevention of a semantic attack requires proper handling of user input. This is similar to the problem of a buffer overflow, which can be handled by hardware tagging. However, there are key differences that are important to understand.

A buffer overflow occurs when there is a violation of the run-time semantics of the programming language. The semantics of the language are usually well understood and standardized. The concept of a buffer is also standard, and exists across many languages. The semantics of the run-time stack and storage of the return address on the stack is intrinsic to the microprocessor. For each microprocessor, it is then possible to develop a mechanism that detects if user data overwrote a return address, or even just overflowed a buffer.

Similar arguments can be made about uninitialized memory, integer overflows or other microprocessor intrinsic run-time semantic violations.

3.7.4.1 Security Tagging and SQL Injection Attacks

For an application-level injection attack, things are a bit different. Lets take SQL injection as an example. If we try to use a hardware-based security tagging approach we need a process that receives user input and tags it – similar to the buffer overflow use case. That user input is then combined with server code to create the SQL query which is then sent to an interpreter. This means that the tag must propagate with the generated query to the interpreter. At some point, probably in the interpretation of the query, security tagging hardware must evaluate the use of the tagged user data and determine if a security exception should be thrown.

What information does the hardware have to make that decision? Instructions are being executed with the user data as operands. A jump or branch to an address that was specified by a user would be a violation, but normal processing of SQL queries would not run into that case. Is there something else? For example is a comparison operation a security violation? Is addition? The answer is that the hardware cannot know the semantic intent of the operation, and there is no fundamental concepts that we can extract from query processing to configure the hardware to generate an error. Therefore, the approach taken in the literature is to cause a security exception every time user data is encountered in an SQL query, and then send the request to a software handler to manage it. The impact of this is:

- There needs to be a software handler that tags user input data. This requirement already exists in most tagging systems, and fits into the model of low-level common functionality.
- There has to be a handler that can differentiate between the processing of an SQL query and processing of the raw input data for filtering purposes. In other words there needs to be software controls to tell the tagging hardware to monitor this processing; which requires the application programmer to be involved in configuring and appropriately using the mechanism.
- Since hardware will not be able to differentiate what part of the query processing is occurring, all queries with user data will cause a security exception. This exception requires a handler to know that the code was processing an SQL query (and not html code for a cross-site scripting attack) and appropriately process the SQL query to

remove errors. This is actually more complex than requiring the user to use a standard SQL query library with security features.

Given the preceding impact it is clear that we have to trust the developer to understand the underlying tagging mechanism and how to tailor it for each type of semantic attack that is a threat to their application. This is just a new flavor on the current problem and does not improve the situation for the developers.

There is a need for further research to determine common elements of high level semantic attacks and common security mechanisms for them. At this point, it appears that the high-level semantic nature of the attacks – they are attacking the functional behavior of the code – is beyond the capability of inherent security mechanisms.

Approved for Public Release; Distribution Unlimited.

4. CONCLUSIONS

The objective of this research was to investigate the design of new operating system architectures in support of the MILS security architecture in the presence of a new security tagged-architecture (STA) microprocessor being developed concurrently by Cornell University research team. The assumption was than an STA-based Operating System (OS) implemented using MILS principles would provide a secure computing foundation that will assist software developers in creating more secure code. To that end we explored the security policy ramifications of STA architectures and tagging for a real-time operating system.

Security Policy Research: Prior work in this area defines run-time solutions as meeting a set of enforceable security polices, which are a subset of all policies. We found that a general purpose STA hardware implementation can only enforce a subset of the run-time enforceable security policies, those based on checks for type mismatches. Specifically, the hardware based STA is only aware of the security tags and associated domains that it supports. This mechanism can be used to support higher level security policies similar to how current microprocessors which are not aware of different users can still be used to support separation between those users.

Conclusions: Hardware-based STA can be used to detect type mismatches in memory access, control flow operations and machine code operations. The assignment of types to memory addresses and registers must be under the control of software. Therefore additional software, in the operating system or even at the middleware and application level, is needed to set the tags, and interpret errors generated by the hardware. This software will be more complex than traditional operating system memory protection software and will therefore require increased verification and validation.

STA hardware provides continual checks for type mismatches, relieving software of that burden, and providing greater confidence in the correct behavior of the system. However, care must be taken to not assume more functionality in the STA hardware than really exists. Additional work is needed to understand the tradeoffs between STA supported security features and software-only based security features.

In addition, in order to provide strong assurance of run-time enforceable security policies we contend that all executable hardware of the system (Direct Memory Access (DMA) controllers, co-processors, network cards, etc.) will have to conform to STA principles, or will have to be isolated by STA hardware; a future area of research.

Security Tagging for a Real-Time Operating System: The second part of this project involved research into operating system architectures and architectural support for STA. This involved development of a new tagging scheme and associated security policy, simulation and testing of that tagging scheme, and implementation of an operating system prototype that utilizes the tagging scheme.

It was decided to utilize the RTMES operating system as a basis for the STA-based operating system instead of starting from scratch. Our original focus was on system architecture and theory but we found that working with an operational system allowed us to investigate our solution on a real system, encountering problems that are often abstracted away at the theoretical and architectural levels. Utilizing RTEMS also enabled us to avoid much of the lower level intricacies of a real system and focus on the security aspects of operating system core components.

Conclusions: It is possible to use an STA to provide increased memory protection, separation and isolation in the operating system and among system applications. Enhanced tags can be used to type memory in terms of function entry points, executable code and data. In addition, the tags can be used to limit control flow between different functional units, and limit access to sets of code modules even in the same address space. This is fully in line with the MILS security architecture concept.

Future Work: Additional work needs to be conducted to determine the best mapping of STA features and tag data types to operating system and application needs. A major drawback of enhanced tagging is the increased overhead of maintaining the tags in memory, and accesses to that memory concurrently with system data.

Approved for Public Release; Distribution Unlimited.

5. REFERENCES

[Alves-Foss06] J. Alves-Foss, W. S. Harrison, P. Oman, and C. Taylor. The MIL} architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2:239-247, 2006.

[Anderson72] J. Anderson, Computer Security Technology Planning Study, ESD-TR-73-51, ESD/AFSC, Hanscom AFB, Bedford, MA 01731, October 1972.

[Bell75] D. Bell and L. LaPadula. Secure computer system unified exposition and Multics interpretation. Technical Report MTR-2997, MITRE Corp., Bedford, MA, July 1975.

[Biba77] K. J. Biba. Integrity considerations for secure computer systems. MTR-3153, Rev. 1, The Mitre Corporation, 1977.

[Dalton07] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of the 34th annual international symposium on Computer architecture*, volume 35, pages 482-493, May 2007.

[Denning82] D. E. R. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[SISm] European Space Research and Technology Center (ESTEC). *{SPARC* instruction set simulator manual}, version 3.0.5 edition, August 2006.

[Fen74] J. S. Fenton. Memoryless subsystems. *The computer journal*, 17(2), 1974.

[Feustel73] E. A. Feustel. On the advantages of tagged architecture. *Transactions on Computers*, C-22(7):644-656, July 1973.

[SPARC] S. I. Inc. *The SPARC Architecture Manual: Version 8.* Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[Kannan09] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 105-114, Estoril, Lisbon, Portugal, 2009. IEEE.

[Kaufmann00] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[Libicki95] M.C. Libicki. *What is information warfare?* National Defense University, Institute for National Strategic Studies, Washington D.C., 1995.

Approved for Public Release; Distribution Unlimited.

[Ligatti05] J. Ligatti, L. Bauer, D. Walker: Edit automata: enforcement mechanisms for run-time security policies. *Int. J. Inf. Sec*. 4(1-2): 2-16, 2005.

[OWASP10] OWASP, *OWASP Top Ten Project* [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[PCMag] P.C. Magazine, *Definition of: semantic attack*.' [Online]. Available: http://www.pcmag.com/encyclopedia/term/51087/semantic-attack

[Rtems] On-Line Applications Research Corporation. *RTEMS C User's Guide*, edition 4.10.1, for RTEMS 4.10.1 edition, July 2011.

[Qin06] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006)*, pages 135-148. IEEE Computer Society, 2006.

[Saltzer75] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(19):1278-1308, Sept. 1975.

[Schneider00] F.B. Schneider, Enforceable security policies. *ACM Trans. Inf. Syst. Secur*., 3(1):30-50, Feb. 2000.

[Schneier00] B. Schneier, Inside risks: Semantic network attacks. *Communications of the ACM*, 43(12):168, Dec. 2000.

[Shioya09] R. Shioya, D. Kim, K. Horio, M. Goshima, and S. Sakai. Low-overhead architecture for security tag. In *15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 135-142, Shanghai, China, 2009. IEEE Computer Society.

[Shriraman10] Shriraman and S. Dwarkadas. Sentry: Light-weight auxiliary memory access control. In *Proc. 37th International Symposium on Computer Architecture 37th (ISCA'10)*, pages 407-418, Saint-Malo, France, June 2010. ACM SIGARCH.

[Shrobe09] H. Shrobe, A. DeHon, and T. Knight. Trust-management, intrusion tolerance, accountability, and reconstitution architecture (TIARA). Technical report, AFRL Technical Report AFRL-RI-RS-TR-2009-271, December 2009.

[Song12] J. Song. *Development and evaluation of a security tagging scheme for a real-time zero operating system kernel*. Master project, University of Idaho, May 2012.

[Song13a] J. Song and J. Alves-Foss. Security tagging for a zero-kernel operating system. In *46th Hawaii International Conference on System Sciences (HICSS)*, pages 5049-5058, Wailea, HI, USA, Jan. 2013.

[Song13b] J. Song and J. Alves-Foss. Hardware security tags for enhanced operating system security. In *IACIS 2013 International Conference*, San Juan, Puerto Rico, USA, Oct. 2013.

[Suh04] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. 11th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85-96, Boston, MA, USA, Nov 2004.

[Witchel02] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 304-316, 2002.

[Yong03] S. H. Yong and S. Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference*, pages 307-316, Helsinki, Finland, Sep, 2003. ACM.

[Zeldovich08] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis. Hardware enforcement of application security policies using tagged memory. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, Proceedings*, pages 225-240, San Diego, California, USA, December 2008. USENIX Association.

Approved for Public Release; Distribution Unlimited.

Appendix A. API for Tag Manager and User Manager

We have developed a tagging scheme for an operating system that utilizes features of a security tagged architecture microprocessor being developed as part of this larger research project. To support the operating system level tagging, we need to have a tag manager in the system. The purpose of the tag manager is to allow system initialization software to configure the initial tags of the system, to allow trusted software to set and modify tags, and to manage the tagging exceptions thrown by the hardware when a tag violation occurs. This section outlines the directives of the tag manager for the RTEMS system. Implementation of this manager is left for future work, waiting for completion of the supporting hardware. The directives of the tag manager are:

- **rtems_tag_partition** - tag every word in an RTEMS memory partition
- **rtems_tag_segment** - tag every word in an RTEMS memory segment of a specified memory region
- **rtems_tag_word** - tag a specified memory word
- **rtems_tag_copy** - copy the value and the tag, set the copy-bit of the tag
- **rtems_tag_validate_copybit** - check the copy-bit of a tag
- **rtems_tag_release** - downgrade the tag to PC tag
- **rtems_tag_enable** - enable the tagging mechanism (this may end up being a startup function that is part of the initialization routines).
- **rtems_tag_disable** - disable the tagging mechanism (a dangerous function, and maybe restricted to just the shutdown phases of RTEMS).
- **rtems_tag_set_handler** - set the exception handling function -- this is for user-level security exceptions that are not already processed by the tag manager
- **rtems_tag_set_lattice** - set rules to form a lattice based on both of the Owner field and the Code-space field in the tag
- **rtems_tag_initialize** - initialize the tag manager

TAG_PARTITION - Tag an RTEMS memory partition

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_partition(
      rtems_id  id,
      rtems_tag tag
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - partition tagged successfully
RTEMS_INVALID_ID - invalid partition id
RTEMS_INVALID_TAG - invalid security tag
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to set the security tags of every word of memory within the specified memory partition.

Approved for Public Release; Distribution Unlimited.

TAG_SEGMENT - Tag a segment of an RTEMS memory region

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_segment(
   rtems_id  id,
   void *segment,
   rtems_tag tag
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - segment tagged successfully
RTEMS_INVALID_ID - invalid region id
RTEMS_INVALID_ADDRESS - segment is null
RTEMS_INVALID_TAG - invalid security tag
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to set the security tags of every word of memory within the specified segment of a memory region.

Approved for Public Release; Distribution Unlimited.

TAG_WORD - Tag a word of memory

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_word(
   int32   *word,
   rtems_tag tag
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - word tagged successfully
RTEMS_INVALID_ADDRESS - word is null
RTEMS_INVALID_TAG - invalid security tag
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to set the security tag of the specified word of memory.

Approved for Public Release; Distribution Unlimited.

TAG_COPY - Set the copy bit and tag of a word of memory

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_copy(
   int32 value,
   int32   *word
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - copy-bit set successfully
RTEMS_INVALID_ADDRESS - word is null
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to set the copy-bit in the security tag of the specified word of memory, while copying a tag and data value. This bit is used to allow a user function to possess a tagged identifier for future use by directives, but to ensure that it cannot be modified.

Approved for Public Release; Distribution Unlimited.

TAG_VALIDATE_COPYBIT - Check the copy-bit of a tag

CALLING SEQUENCE:

```
     rtems_status_code rtems_tag_validate_copy-bit(
            rtems_tag tag
     )
```

DIRECTIVE STATUS CODES:

RTEMS_COPYBIT_NOT_SET - the copy-bit is not set
RTEMS_COPYBIT_SET - the copy-bit is set
RTEMS_INVALID_TAG - invalid security tag

DESCRIPTION

This directive checks the copy-bit of the tag.

Approved for Public Release; Distribution Unlimited.

TAG_RELEASE - Downgrade the tag to PC tag

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_word(
   int32   *word,
   rtems_tag PCtag
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - word tagged successfully
RTEMS_INVALID_ADDRESS - word is null
RTEMS_INVALID_TAG - invalid security tag
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive downgrades the security tag of the specified word of memory to the PC's tag.

Approved for Public Release; Distribution Unlimited.

TAG_ENABLE - Turn on the tagging subsystem of the hardware

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_enable(
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - system enabled successfully
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to turn on the tagging subsystem. This may end up being a startup function that is part of the initialization routine.

TAG_DISABLE - Turn off the tagging subsystem of the hardware

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_disable(
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - system disabled successfully
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION:

This directive allows the calling program to turn off the tagging subsystem. This is a dangerous directive and may end up being a startup function that is part of the shut-down routine.

Approved for Public Release; Distribution Unlimited.

TAG_SET_HANDLER - Set the user-level security exception handler

CALLING SEQUENCE:

```
rtems_status_code rtems_tag_set_handler(
   void *handler()
)
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - handler bit set successfully
RTEMS_INVALID_ADDRESS - invalid function pointer
RTEMS_UNAUTHORIZED_ACCESS - security permissions will not permit the operation

DESCRIPTION

This directive allows the calling program to set the user-level exception handler for tagging exceptions not handled by the tag manager.

Approved for Public Release; Distribution Unlimited.

TAG_SET_LATTICE - Set the nodes and relations in the lattice

CALLING SEQUENCE:

```
     rtems_status_code rtems_tag_set_lattice(
        rtems_tag tag1,
        rtems_tag tag2
     )
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL - lattice set successfully
RTEMS_INVALID_TAG - invalid security tag
RTEMS_INVALID_PARAMETER - requested relationship inconsistent with current lattice

DESCRIPTION:

This directive sets rules to form a lattice based on both of the Owner field and the Code-space field in the tag.

Approved for Public Release; Distribution Unlimited.

TAG_INITIALIZE - Tag RTEMS

CALLING SEQUENCE:

```
     rtems_status_code rtems_tag_initialize(
     )
```

DIRECTIVE STATUS CODES:

RTEMS_SUCCESSFUL **-** RTEMS tagged successfully
RTEMS_INVALID_TAG **-** invalid security tag

DESCRIPTION:

This directive initializes the tag manager.

Approved for Public Release; Distribution Unlimited.

LIST OF SYMBOLS, ABBREVIATIONS AND ACRONYMS

ASR	Asynchronous Signal Routine
BA	Branch Always
BN	Branch Never
BSP	Board Support Package
CPU	Central Processing Unit
CSRF	Cross-Site Request Forgery
CWP	Current Window Pointer
DIFT	Dynamic Information Flow Tracking
DMA	Direct Memory Access
DMM	Data Mark Machine
DOM	Document Object Model
GCC	GNU Compiler Collection
ISR	Interrupt Service Routine
LUB	Least Upper Bound
MILS	Multiple Independent Levels of Security
MMP	Mondrian Memory Protection
nPC	next Program Counter
OS	Operating System
OWASP	Open Web Application Security Project
PC	Program Counter
PCR	Propagation Control Register
PLB	Permission Lookaside Buffer
RISC	Reduced Instruction Set Computer
RTEMS	Real Time Executive for Multiprocessor Systems
SCC	Simple Security Condition
SIC	Simple Integrity Condition
SIS	SPARC instruction Simulator
SQL	Structured Query Language
SPARC	Scalable Processor ARChitecture
STA	Security Tagged-Architecture
TCB	Task Control Block
TCR	Trap Control Register
TIARA	Trust-management, Intrusion-tolerance, Accountability, and Reconstitution Architecture
TMU	Tag Management Unit
URL	Uniform Resource Locator
XXS	Cross-Site Scripting
ZKOS	Zero-Kernel Operating System